Support Software

for

Computer Based Learning Materials

Alfred Bork
Augusto Chioccariello
Stephen D. Franklin

Information and Computer Science
University of California
Irvine, California 92717

Technical Report 238

Support Software
for
Computer Based Learning Materials

Abstract

This report presents the principal software tools developed in the Educational Technology Center at the University of California, Irvine to facilitate the implementation of computer based learning materials. The pedagogical specifications for such materials are developed by teams of educational experts and then turned over to coders for implementation using these software tools.

This software provides capabilities in three general areas:

- Window oriented text and graphics display facilities coupled input and timing capabilities.
- File structures and access mechanisms which support separation of the algorithmic content of a program from data needed by the program.
- Facilities for processing, recognizing and classifying responses given by learners to the questions presented by computer based learning materials.

The implementation is done in UCSD Pascal which has a "units" capability similar to "packages" in Ada and "modules" in Modula-2.

Introduction

The Educational Technology Center is a research and development group concerned with the use of modern technology, particularly computers, to aid learning. Over a sixteen year period, we have developed a full scale production process for computer based learning materials.

This process follows the classic life cycle model for software development and parallels that found in the design and production of high quality educational materials in other media such as text and video. Key features of this process are that it separates pedagogical and technical issues and that different teams of specialists participate in the various stages of the process. In particular, educational experts devote their full attention to pedagogical issues, leaving technical concerns to others.

The pedagogical design issues and how this process addresses them are discussed elsewhere (cf., A. Bork, *Personal Computers for Education*, Harper & Row, 1985). This report presents the principal software tools we have developed to facilitate the implementation of sound educational design. This software is intended for use by coders not by the pedagogical designers. Technical work such as these tools facilitate is best left to competent coders. Authoring languages and systems which focus on providing programming tools or environments to educational experts divert the attention of these experts from issues of pedagogical design to technical "can I do this?" or "how do I do this?" questions.

This report is presented in response to inquiries about the software techniques we use in producing computer based learning materials. For us, however, these tools have never been important as ends in themselves. We feel that such technical tools must be evaluated in terms of the educational software whose production they have facilitated. Many of their limitations are deliberate, meant to encourage a particular style of coding or because there was no practical need for anything more general.

Goals of the Software Tools

1. These procedures are designed to help the coder to work more efficiently. The coding should be done as rapidly and as error free as is reasonably possible.

2. Since most of our coders are undergraduates, often with only one or two courses in programming when they begin, the procedures must be relatively easy to use. At the same time, the software has to be flexible and allow coders to achieve complex effects in a simple fashion. Simple things should be easy; complex things possible.

3. As with all good programming the code must be readable. This is particularly important because over a period of time a number of different people may work on the materials and one must always be able to modify it based on experience gained by observing how learners react to it.

4. The code must be easy to transport to other machines. Rapidly developing technology makes it likely that computer based learning materials will need to be moved several times during their lifetimes. This may include changing the programs to take advantage of new capabilities not available when the material was first developed.

Overview of the Software

The software described in this document falls into three general categories, each described in its own section. The first section is concerned with input and output on the screen, both text and graphics. The second describes a type of file, including tools for creating and accessing such files, designed to support separation of the algorithmic content of a program from the data used by the program. The third section covers a collection of software tools which supports the answer processing, commonly needed for computer based learning material.

1. Ports -- Screen Control, Input and Output.

Two important aspects of interactive computer-based educational materials are the manner of displaying text (including spacing and timing) and the way in which textual responses are obtained from the user. The display routines must allow flexibility in positioning and formatting text to enhance readability. The routines for soliciting and accepting user input should be "friendly" to the user and simple for the programmer. An integral part of any software providing these facility must be tools which control timing of the display and the time allowed for the user's responses.

Another important aspect of interactive learning modules is the use of graphics. One needs a system of routines and data structures to simplify the use of graphics. Since the details of graphics vary greatly between devices, such a system must do as much as possible to free the programmer from concern about the particular device being used.

2. Keyed Files -- Separating Data from Algorithms.

The file system we have developed, referred to as keyed files, allows the programmer to remove from a program most of the data it needs (other than that supplied interactively by the user), and to keep it instead in files separate from those that contain code expressing the logic of the program. This separation reduces the size of the program (source and code), and facilitates modification of the program. In particular, the use of keyed files makes it easier to modify or vary the messages presented to the user, to add or change strings that are recognized by the program in users' responses, and greatly facilitates translating the entire program from one natural language to another. It even allows a single program to run in different natural languages simply by using different data files.

3. String Analysis -- Analyzing Learners' Responses.

We use string matching to determine whether the input string, representing the learner's response to a question, matches an expected response as specified by the pedagogical design. The StringAnalysis unit supplies routines and data structure for pattern matching. Rules used to analyze the answer to a given question can be stored in a keyed file under a meaningful name and retrieved by the answer analysis unit.

Implementation

The production of software to assist learning is a complex activity which requires modern software engineering practice for large scale programming. The programming environment for the coding of computer based material should support a structured language with strong type checking, string manipulation and separate compilation which maintains the type checking. The run time environment should provide memory management, random access files and I/O capabilities which allows a program full control of the interaction with the user.

Since 1979, the Educational Technology Center has used UCSD Pascal (an extension of Pascal commercially available on most inexpensive stand alone systems) in its software development. Other computer languages such as Ada(tm) or Modula-2 meet the requirements outlined above. The software tools described in this document are currently implemented as "units" in UCSD Pascal. A "unit" in UCSD Pascal is approximately equivalent to a "package" in Ada and a "module" in Modula-2.

Acknowledgements

Our excellent undergraduate coders are responsible for most of the implementation and have contributed to the design as well. Individual contributions are given at the start of each section, but the following deserve special mention: Stephen Bartlett, Adam Beneschan, Christi Genung, Martin Katz, Alastair Milne, Naomi Salvadori, Tim Shimeall. As project mangers, Barry Kurtz and David Trowbridge made valuable contributions to the design.

Cooperative efforts with our colleagues at the University of Geneva have been very important. We have worked together on projects using this software, they have used it extensively on projects of their own and their contributions to its evolution have been invaluable. We particularly wish to thank Professor Bernard Levrat and Bertrand Ibrahim.

Finally we, as almost everyone using Pascal on personal computers, owe a debt of gratitude to Professor Kenneth Bowles of the University of California San Diego, who headed the project which developed UCSD Pascal.

Ports Unit
Reference Guide
Version 2.0
16 July 1985

This software was developed by the Educational Technology Center of the University of California at Irvine supported, in part, by various federal grants. The initial work was funded by NSF grant #SER78-06471 to the University of California Irvine as part of the NSF CAUSE program.

Address comments or questions to

Alfred Bork or
Augusto Chioccariello
Educational Technology Center
University of California
Irvine, California 92717
(714) 856-6945

or

Stephen D. Franklin
Computing Facility
University of California
Irvine, California 92717
(714) 856-5154

The Ports unit is a fusion of two previously separate units:
TextPorts, designed, implemented and maintained by S. Bartlett, S. Franklin, M. Katz, S. Shimeall, T. Shimeall, and J. Zarbock;
GraphPorts, designed, implemented and maintained by S. Franklin, M. Katz, A. Milne, G. Roberts.

Many people at ETC have contributed ideas toward fusing TextPorts and GraphPorts into a single unit. The final design and implementation is principally the work of S. Bartlett, A. Chioccariello, S. Franklin, and D. Trowbridge.

Version 2.0 introduces color into Ports, a capability not used in previous versions. The design and implementation is principally the work of A. Chioccariello, S. Franklin, and C. Genung.

We were greatly assisted by the thoughtful comments by our colleagues at the University of Geneva, Professor Bernard C. Levrat and Bertrand Ibrahim.

The Ports unit is implemented in Pascal using the UCSD p-System (tm) developed at the University of California San Diego's Institute for Information Systems, under the direction of Professor Kenneth L. Bowles. The UCSD p-System is now maintained and distributed by SofTech Microsystems.

Table of Contents

1. Introduction

Two of the most important aspects of interactive computer-based educational materials are the manner of displaying text and the way in which textual responses are obtained from the user. The display routines must allow flexibility in positioning and formatting text to enhance readability. The routines for accepting and processing user input should be "friendly" to the user and simple for the programmer. An integral part of any software providing these facilities must be tools which control timing of the display and the time allowed for the user's responses.

Another important aspect of interactive educational materials is the use of graphics. Thus, we need a system of routines and data structures to simplify the use of graphics. Since the details of graphics vary greatly between devices, such a system must do as much as possible to free the programmer from concern about the particular device(s) being used.

"Ports" is a UCSD Pascal unit designed to meet the needs just outlined. It is a collection of routines and data structures which support text and graphics in interactive programs on small stand-alone systems. The current unit is a fusion of two previous ones, TxtPort and GrphPort, which have been essential in the development of interactive educational materials at ETC over the past several years.

This document provides a technical description of the Ports unit for people coding or reading programs which use it. Additional documentation is available containing implementation information necessary only to those who are working on the Ports unit itself.

The fundamental concept underlying this unit is that of the port: a rectangular region of the computer screen capable of displaying both textual and graphical information. A port may be treated as a single, inseparable entity, or as the combination of two components: one dealing with text only, and the other dealing with graphics only. These components are termed "textport" and "graphport," respectively.

The Ports unit contains four kinds of routines, constants, types, and variables. The first kind treats a port as a single entity and their names are generally prefixed by "Pt" (e.g. "PtDefine"). The second kind references only the textual component; these names are generally prefixed by "Txp." The third kind deal solely with the graphical component; they are generally prefixed by "Gr." Finally, some routines do not deal with ports at all, so their names generally have no prefix (e.g. "TimerSet," "ChainTo"). For historic reasons having to do with the evolution of Ports and the need for backward compatibility, there are exceptions to these naming conventions. The best rule is to keep the general conventions in mind in reading the documentation but to rely on the documentation more than on general conventions.

Most of the routines in the Ports unit must reference all or part of a port in order to function. For some of these routines, the port to access is passed as a parameter. Usually, however, the routine gets its information from a place which depends on the routine's prefix.

Many "Txp" routines refer to and act on the "Currently Active Textport," or CAT, the one port whose textual component is currently "active." Many "Gr" routines access the "Currently Active Graphport," or CAG, the one port whose graphic component is currently "active." There is always exactly one CAT and one CAG. If the CAT and CAG refer to the same port, then this port is also the "currently active port" or CAP. If the CAT and CAG are components of different ports, the CAP is "undefined."

Each port has a color scheme associated with it. A color scheme has such information as a port's background color, drawing colors, and text colors.

For each port, the Ports unit keeps track of two special positions:
1) the Current Text Position (CTP) -- the place in the port where the next text reading or writing operation will occur;
2) the Current Graphics Point (CGP) -- a point referred to by all drawing and moving operations on the port.

In reference to the graphical drawing and moving routines:
All these routines use the CAG. They all update that graphport's CGP. The coordinates of all points in the graphport, including the CGP, are given by the "real world" coordinates which the programmer sets.

The Ports unit AUTOMATICALLY "clips" any portion of a graphical object which doesn't lie within the window, and which therefore cannot be displayed in the graphport.

In general, drawing a new graphical item (a line segment, or an arc, for example) involves drawing from the CGP to a new point and updating the CGP to that point.

The standard reference on interactive computer graphics is the book "Principles of Interactive Computer Graphics" by Newman and Sproull (second edition, published by McGraw-Hill, 1979).

In this document, "user" refers to a person who uses (i.e., runs, executes) any program which employs the Ports unit. The person who writes such programs is called the "programmer" or "coder."

While this document mixes upper and lower case letters in identifiers to increase their (human) intelligibility, the case of any letter in an identifier is semantically irrelevant to Pascal. That is, to Pascal "PORTS", "Ports," and "ports" are the same identifier.

In order to use the Ports unit in a program, the program header must be:
```
PROGRAM Dialog;
USES Ports;
```
This is the simplest form, which assumes that the Ports unit is in the file *SYSTEM.LIBRARY.

2. Global Declarations

Read the following CONST, TYPE and VAR sections carefully. They not only define important constants, types and variables that the Ports unit uses, but they also give identifiers whose use is preempted by their role in the Ports unit.

```
CONST PtLeftArrow  = ORD(Character_code_representing_Left_Arrow_key);
      PtRightArrow = ORD(Character_code_representing_Right_Arrow_key);
      PtUpArrow    = ORD(Character_code_representing_Up_Arrow_key);
      PtDownArrow  = ORD(Character_code_representing_Down_Arrow_key);
      PtScrnBottom = <bottom row number -- 0 = top>
      PtScrnRight  = <right column number -- 0 = left>
      PtCharWidth  = <number of graphic pixels wide a text font is>
      PtCharHeight = <number of graphic pixels high a text font is>
      PtPixAspect  = <ratio of height of one pixel to width of one pixel>
      PtMaxColors  = 3;  { maximum range of colors }

TYPE TxpOption = (LeftAdjust, RightAdjust, Centered,
                  AskScroll, AutoScroll, DemandScroll, NoScroll,
                  AnyCase, UpperCase, LowerCase,
                  SingleSpacing, DoubleSpacing,
                  NoClearLine, ClearLine,
                  NoStartLine, StartLine,
                  NoEndLine, EndLine,
                  Tone, NoTone,
                  Echo, NoEcho,
                  Slow, NoSlow);
     TxpOptSet = SET OF TxpOption;
     TxpLongString = STRING[255];
     PtColorRange = 0 .. PtMaxColors;
     PtCSName =     { Names that can be used for color schemes }
               (PtCSSystem, PtCSHelp, PtCSSummary,
                PtCS1, PtCS2, PtCS3, PtCS4, PtCS5);
     PtTextMode =       { Different ways of writing text in a port }
               (PtEchoed,  { Input from learner which has been echoed }
                PtNormal,  { Normal output }
                PtLoud,    { Emphasized output }
                PtQuiet);  { De-emphasized output }
     TimerType = REAL;
     Alphabet = SET OF 'A'..'Z';
     GrLnMode =       { Types of lines that can be drawn }
        (GrNone,      { Do nothing to pixels in path }
         GrReplace,   { Turn all pixels in path to current line color }
         GrClear,     { Turn all pixels in path to port background color }
         GrInvert);   { For every pixel in path invert the its with the
                        current line color.  Repeating the operation on
                        the same pixel restores the previous color }
     GrLnStyle =      { Styles of lines that can be drawn }
        (GrSolid,     { affects all pixels on the line }
         GrDotted,    { affects every other pixel on the line }
         GrDashed);   { affects every other 2mm (approx.) of
                        pixels on the line }
```

```
VAR ESCOk: BOOLEAN;   { Answers the question "When the ESCKey is
                        pressed, is it OK to honor the user's request for
                        Extra Service Control?"  ESCOk is normally TRUE, but
                        may be set FALSE to disable the handling of the ESCkey
                        by textual input/output routines.  This should be done
                        ONLY in VERY special and unusual circumstances (e.g., a
                        training sequence on the use of the ESCkey).  When ESCOk
                        is FALSE, ANY ESC request terminates the Ports unit input
                        routine being executed when the ESCkey was pressed. }
    TxpDebug: BOOLEAN;  { A programmer debugging aid initialized to
                        FALSE by PtInit [3.1].  May be set TRUE either by the
                        program or at runtime through TxpESC [3.9]. When TRUE,
                        an "infinite" read time is assumed for all input routines
                        (see "Timing Out" [8]).  See section 7 for more information.
    TxpMsgName: STRING[10];  { Part of the debugging message output
                        by TxpESC [3.9].  Initialized to '' (null string) by
                        PtInit.  If the program is using the Keyed File system
                        display unit, TxpMsgName holds the name of the Keyed
                        File message most recently displayed. See section 8. }
    GrRubberBanding: BOOLEAN;  { Normally FALSE indicating that the
                        graphics cursor is simply a small set of cross-hairs.
                        When TRUE it means that the graphics cursor is the small
                        set of cross-hairs PLUS a "rubber band" from the center
                        of the cross-hairs to the Current Graphics Point (CGP). }
```

WARNING:

The programmer is URGED in the STRONGEST terms possible to refrain from directly accessing the internal components of ports as described in the following declarations. They are included ONLY because they cannot be hidden (and because there just may be some circumstances when one MUST know what is "really" happening in Pascal).

```
TYPE TxpSwitches = PACKED RECORD
                     AlignType: LeftAdjust..Centered;
                     ScrollType: AskScroll..NoScroll;
                     CaseType: AnyCase..LowerCase;
                     SpacingType: SingleSpacing..DoubleSpacing;
                     ClearLine, StartLine, EndLine, Tone,
                     Echo, Slow: BOOLEAN;
                   END;

    Port = ^Prt;

    Prt = RECORD
            Left, Top,    { Location of start of the textport in
                            text coordinates.  The text coordinate
                            system has its origin (0, 0) in the
                            upper left corner of the screen, with
                            column and row indices increasing to
                            the right and down. }
            Width, Height,  { Dimension of the textport, minimum
```

```
      width and height of 1 }
    Row, Col,     { Physical position of virtual text cursor
                    within the port.  The coordinate system
                    of the virtual text cursor mimics that
                    of the screen. }
    Old,   { Number of lines scrolled out of top of port
             since it was last erased (0 if none scrolled). }
    IOStatus,     { Error code; =0 if no error }
    ScrollSize: INTEGER;  { Number of lines to be scrolled
                            when port is scrolled }
    Parm: TxpSwitches;  { Default parameters for textport I/O }
    History: SET OF (WndwDefined, PortColored);
      { flags for the port:
              - has the window been defined?
              - is the background of the Port colored? }

    CrScheme: PtCSName; { Name of the color scheme for this port }
    LineMode: GrLnMode; { Current line drawing mode }
    LineStyle: INTEGER; { Current style in internal INTEGER form }
    WriteColor: PtTextMode; { Currently active writing mode }
    LineColor: PtColorRange; { Currently active graphics color }
    XCurrent, YCurrent,
    XScale, YScale: REAL;
    XTranslate, YTranslate: INTEGER;
      { Scaling factors and translation offsets for converting
        window coordinates into screen (pixel) coordinates }
    XMin, YMin, XMax, YMax: INTEGER;
      { Graphport boundaries in screen (pixel) coordinates }
    XCursorStep, YCursorStep: INTEGER;
      { Step distance for each graphic cursor movement }
    Next: Port; { Used internally for putting ports in lists }
END;
```

The actions of some of the Ports timing routines are best described in terms of certain variables which are internal to the Ports unit (and therefore not accessible to programs using the unit):

TxpTicToc: INTEGER; Factor for output timing and pauses.

Limits, in seconds, for timing TxpRead:

TxpReadTime: INTEGER; Total time allowed.

TxpStrokeTime: INTEGER; Time allowed for next keystroke.

3. Routines

The following abbreviations (all, alas, beginning with "C") are used extensively in describing the routines of the Ports unit:

```
CAG -- Currently Active Graphport
CAP -- Currently Active Port
CAT -- Currently Active Textport
CGP -- Current Graphics Point
CTP -- Current Text Position
Note that "C" means "Current" or "Currently,"
          "A" means "Active,"
          "G" means "Graphport" or "Graphics,"
          "T" means "Textport" or "Text,"
          "P" means "Port," "Point" or "Position."
```

3.1 Ports Management

OVERVIEW

Before using any other Ports routines, a program must first initialize any color schemes (one of which must be PtCSSystem [3.2]), and then initialize the Ports system by PtInit, once and only once.

After the color schemes and the Ports system have been initialized, the typical sequence of procedure calls in using a port is as follows:

- Use PtSetCrScheme to select the current color scheme for the port(s) that will be defined (unless the default color scheme or the current color scheme will be used).
- Define the port using PtDefine; this MUST be done BEFORE anything else is done with the port.
- Select the port using PtSelect, TxpSelect or GrSelect.
- If the PtNormal text mode will not be used for the current writing color, set the text attributes with TxpSetWrColors.
- Window the port using GrWindow if the port is going to be used for graphics other than framing (GrFrame) and clearing (PtErase).
- Set the port's graphics attributes with GrSetLnMode, GrSetLnStyle, GrStepSize, and GrSetLnColor if the port will be used for graphics and the default graphics attributes are not appropriate.
- Use the port, selecting other ports when appropriate, reselecting this one, and resetting graphics attributes or text options (with TxpNewOptions) as needed.
- Dispose of the port with PtDispose when the port as defined is no longer needed.

Each section of the program should define (and later dispose of) as many ports as the logic of the section dictates and no more.

IMPORTANT NOTE:

If the program logic requires that a port be redefined, the program should dispose of the port (using PtDispose) BEFORE calling PtDefine for the port another time.

```
PROCEDURE PtInit;
```

PtInit MUST be called AFTER the color scheme PtCSSystem [3.2] is defined and BEFORE anything else in the Ports unit is used. Generally, it should be the first executable statement and the color schemes are defined in any program using Ports. PtInit does a number of things without which the Ports unit will not even begin to work properly. Any further calls to PtInit in a program will dispose all ports except the system message port and clear the screen to its background color.

```
PROCEDURE PtDefine(VAR Name: Port; Left, Top, Width, Height: REAL;
                   ScrollDepth: INTEGER; Options: TxpOptSet);
```

PtDefine defines the new port Name so that
1) its upper left corner is
   Left text columns (character widths) from the screen's left edge and Top text rows (character heights) from the screen's top edge,
2) the port is Width text columns wide and Height text rows high,
3) its default textport scroll size is set to ScrollDepth,
4) its default textport options are specified by the set Options, and
5) its color scheme name is set to the current color scheme name.

Note that this specification means that the leftmost text column on the screen is column 0 and the top row is row 0.

In addition:

PtDefine sets the port's IOStatus to 0. (The value of IOStatus may be read via the TxpStatus function [3.5]. The meanings of these values and the routines which alter it are documented in [5].),

PtDefine sets the following defaults for graphics:
WriteColor is set to PtNormal (may be altered by TxpSetWrColors),
LineColor is set to GraphicsColor[1] (may be altered by GrSetLnColor),
LineMode is set to GrReplace (may be altered by GrSetLnMode),
LineStyle is set to GrSolid (may be altered by GrSetLnStyle),
Cursor step sizes are set to minimum motion in each direction (may be altered by GrStepSize).

A port MUST be defined (using PtDefine) BEFORE any other routine is called which attempts to use it in ANY fashion!

Before any graphics can be displayed in the port (other than framing or erasing), the port's "windowing" must be established by selecting the port (using either PtSelect or GrSelect) and specifying the window parameters (using GrWindow). Once this is done, graphics can be displayed anywhere in the area specified by the PtDefine parameters.

Text can be displayed ONLY at fixed positions on the screen (a whole number of character widths from the left edge and a whole number of character heights from the top). If all the REAL parameters in the call to PtDefine represent whole numbers, the area in which text can be displayed will be exactly the same as that in which graphics can be displayed. If any of these REAL parameters are not integers, the area in which text can be displayed may be somewhat SMALLER than that specified by the PtDefine parameters. In judging whether a real number is an integer, PtDefine allows a margin of error of about 0.01.

To be more precise:
The first column in which text can appear is TRUNC(Left+0.99);
the last column is TRUNC(Left+Width-0.99).
The first row in which text can appear is TRUNC(Top+0.99);
the last row is TRUNC(Top+Height-0.99).
EXAMPLE:
Given the statement
PtDefine(Reply, 1.8, 4.1, 12.2, 3.8, 1, []),
the text display area has (2,5) as its upper left corner,
is 12 columns wide and 2 rows high.

The port area specified by the PtDefine parameters MUST include at least one full character display position. Thus, the statement
PtDefine(Reply, 1.1, 2, 1.8, 3, 1, [])
is illegal because its width (from 1.1 to 2.9) includes part of 2 text columns but not a whole one.

Each time PtDefine is called, the system allocates to the port the memory it requires. When the program is finished using the port as defined, it should call PtDispose to release that memory. Calling PtDefine for the same port again, WITHOUT having called PtDispose in the meanwhile, irretrievably loses the memory allocated by the previous call to PtDefine!

PROCEDURE PtDispose(VAR Name: Port);

PtDispose "undefines" the named port, releasing for other use the memory allocated to it by PtDefine. It also "disappears" the port by filling it with the screen background color if it has not already been done. PtDispose should be called whenever there is no further need for the port as currently defined (i.e., we don't need the port at all or it needs to be located elsewhere on the screen). Once a port has been disposed of, it can not be used until it is defined again using PtDefine.

PROCEDURE PtSelect(Name: Port);

PtSelect selects the port Name as the currently active port (CAP), making it also the currently active textport (CAT) and the currently active graphport (CAG) as well. It colors the port to its background color if it has not previously been done. All textual input and output is done through the currently active textport (CAT); all graphical I/O is done through the currently active graphport (CAG). The CAT and CAG may be changed as often as needed, either separately (via TxpSelect or GrSelect) or jointly (via PtSelect). Once a port is selected as CAP, CAT or CAG via these routines, it stays selected until another port is selected by a subsequent call to them.

PROCEDURE TxpSelect(Name: Port);

TxpSelect selects Name as the CAT, positions the text cursor at the CTP in Name, and colors the port to its background color if it has not previously been done. Thus, Name is the port referred to

by all routines which access or modify the CAT. Name remains the CAT until a subsequent call to TxpSelect or PtSelect designates another port as the CAT.

PROCEDURE GrSelect(GrPort: Port);

GrSelect selects GrPort as the currently active graphport (CAG) and colors the port to its background color if it has not previously been done. All graphic input and output uses this graphport until another one is selected by a subsequent call to GrSelect or PtSelect.

PROCEDURE TxpNewOptions(NewOptions:TxpOptSet);

This procedure sets the default options of the CAT to NewOptions, without changing anything else about the port.

PROCEDURE GrWindow(Left,Bottom,Right,Top:REAL);

In terms of graphics, each port provides a way of viewing some portion of an external "real world." The programmer is free to choose whatever coordinate system seems most "natural" (i.e., easiest to use) for this "real world." Thus, "real world coordinates" can also be thought of as "programmer chosen and convenient coordinates."

GrWindow associates the "real world" coordinate system convenient for the programmer and the actual area on the screen occupied by the CAG. This association is given by the GrWindow parameters as follows:
Left is the "real world" X-coordinate of the CAG's left edge,
Bottom is the "real world" Y-coordinate of the CAG's bottom edge,
Right is the "real world" X-coordinate of the CAG's right edge,
Top is the "real world" Y-coordinate of the CAG's top edge.
Technically speaking, these four values specify a Cartesian coordinate system to be associated with the port which is the CAG at the time GrWindow is invoked. This coordinate system remains attached to the particular port, even when it is no longer the CAG. Thus, after a port has been selected and "windowed" ONCE, it DOES NOT NEED to be "windowed" every time it is selected.

In addition, GrWindow sets the current graphic point of the port to (0,0) and sets the graphic cursor step size to minimum movement in both X and Y directions (this may be altered by GrStepSize).

The window may be defined with any orientation of coordinates:
one may have Left < Right or Right < Left;
one may have Bottom < Top or Top < Bottom.
There is special provision (see below) for Left = Right or Bottom = Top.
The only case that is NOT allowed is the "zero area window":
Left = Right AND (in the same GrWindow) Bottom = Top is illegal.
For example, GrWindow(5,7,5,7) is illegal.

SPECIAL CASE of Left = Right or Bottom = Top (but NOT BOTH):

The window specified has the same scale in both the horizontal and the vertical direction; that is, on the screen, 1 cm. horizontally and 1 cm. vertically represent the same number of "real world" (programmer-defined) units.
This scale is specified by the non-coinciding pair of edge coordinates (Left and Right or Top and Bottom).
The common value of the pair of edge coordinates that coincide is taken as the coordinate of the center of the window.
EXAMPLE:
"GrWindow(-3,7,5,7)" says that the "currently active graphport" is to look at a window on the "real world" which goes from -3 on the left to 5 on the right and whose center point has coordinates ((-3+5)/2,7)=(1,7).
In this window, 1 "real world" unit in the horizontal (X) direction and 1 "real world" unit in the vertical (Y) direction have the same length on the screen.
This means that the graph of a "real world" circle will look like a circle and not like an oval.

NOTES:
If a port is going to be used for any graphics other than framing (GrFrame) or clearing (GrErase), it must be "windowed" (using GrWindow) each time after it is defined.
Although GrWindow refers to the CAG, the windowing information given by its parameters is associated with the port which is the CAG and stays with that port even when another port is selected as the CAG.

EXAMPLE:

```
PtDefCS(PtCSSystem, 'CRWKCWCRRCCK');  { MUST come before calling PtInit }
PtInit;  { MUST come before calling any other Ports unit routines }
PtDefine(Picture, 20, 5, 20, 10, 1, []);
PtDefine(Words, 50, 10, 20, 5, 3, [RightAdjust]);
PtDefine(Read, 0, 1, 20, 3, 2, [NoSlow]);
PtSelect(Picture);  { Picture is CAP, CAT, and CAG }
GrWindow(-300, -300, 300, 300);  { Define coordinate system for Picture
...  { statements doing text and graphics I/O in Picture}
TxpSelect(Read);  { Read is CAT; Picture is still CAG; CAP is not defined
...  { statements doing text I/O in Read and graphics I/O in Picture }
TxpSelect(Words);  { Words is CAT; Picture is CAG; CAP is not defined }
...  { statements doing text I/O in Words and graphics I/O in Picture }
GrWindow(-30, 0, 30, 0);   { Define new coordinate system for Picture }
...  { statements doing text I/O in Read and graphics I/O in Picture }
TxpSelect(Picture);  { Picture is now CAT and CAG; thus, it is also CAP }
TxpNewOptions([Centered]);  { Change default text options for Picture }
...  { statements doing text and graphics I/O in Picture}
PtDispose(Read);  { We are done with Read }
PtDispose(Words);  { We are done with Words in its current position }
PtDefine(Words, 45, 5, 25, 15, 10, []);  { but do want Words elsewhere }
TxpSelect(Words);  { Words is CAT; Picture is CAG; CAP is not defined }
...  { statements doing text I/O in Words and graphics I/O in Picture }
PtDispose(Words);  { We are done with Words }
PtDispose(Picture);  { We are done with Picture }
```

3.2 Color

Encoding of Color Schemes

To facilitate the definition of color schemes, each color is represented by a character. In most cases this character is the first letter of the color's name. The color encoding scheme is as follows:

```
W - White                C - Cyan
Y - Yellow               G - Green
M - Magenta              B - Blue
R - Red                  K - blacK
```

There are 12 components of a color scheme. The first 4 specify the background and graphics colors, the last 8 give the text writing colors. The graphics colors are numbered 0 to 3, with color 0 also specifying the port background color and colors 1 through 3 being other drawing colors. Thus, a color scheme can be encoded by a 12 character string in the following manner:

Character	Representation
1	graphics color 0 = port background color
2	graphics color 1
3	graphics color 2
4	graphics color 3
5	PtEchoed background
6	PtEchoed foreground
7	PtNormal background
8	PtNormal foreground
9	PtLoud background
10	PtLoud foreground
11	PtQuiet background
12	PtQuiet foreground

```
PROCEDURE PtDefCS(Name: PtCSName; CS: STRING);
```

PtDefCS defines the color scheme identified by Name according to the string CS, which is encoded as described above. The color scheme identified by Name MUST be defined using this procedure before Name can be used anywhere else. PtCSName is an enumerated type which lists all values allowed for Name.

```
PROCEDURE PtSetCrScheme(NewScheme: PtCSName);
```

PtSetCrScheme sets the current color scheme to NewScheme, which is a name of a color scheme that has previously been associated with a color scheme via PtDefCS. Any port defined after setting the current color scheme to NewScheme will use NewScheme. PtInit initializes the default color scheme to PtCSSystem. Changing the current color scheme does not affect ports already defined.

PROCEDURE TxpSetWrColors(NewMode: PtTextMode);

TxpSetWrColors sets the current writing color (or font) of the CAT to NewMode. NewMode is the name of a writing mode (i.e., PtEchoed, PtNormal, PtLoud, or PtQuiet). Subsequent writing in this port is done using these colors. Only the CAT is affected, and this will not be apparent until the next time text is written in this port.

PROCEDURE GrSetLnColor(NewLnColor: PtColorRange);

GrSetLnColor sets the line color of the CAG to NewLnColor. Subsequent drawing in this port is done using this color. Only the CAG is affected, and this will not be apparent until the next graphics output.

Defining Palettes and Color Schemes

The following section describes how a palette and color schemes are defined. A palette is a set of four colors which may be used at any one time. Thus a program normally has one palette of four colors, and a set of color schemes. Each color scheme is a combination of the four colors of the palette that defines the colors of the background, text, and graphics of a port.

NOTE: Everything described in the following section MUST be done BEFORE PtInit!. PtInit needs the information about the palette before it can even begin to work properly at all, let alone in color. Thus, PtInit is NOT the first Ports routine that is called in a program that uses Ports.

Each color scheme has a color scheme name associated with it. When a color scheme is used, it is referred to by its name. There are eight predefined color scheme names in Ports that are available: PtCSSystem, PtCSSummary, PtCSHelp, PtCS1, PtCS2, PtCS3, PtCS4, and PtCS5. The programmer may associate any one of these names with a color scheme; the association between the name and the color scheme should help the user identify the purpose of the color scheme. For example, if a particular color scheme is to be used when the user is having problems running a module, the color scheme could have the name PtHelp.

The procedure PtDefCS has two purposes. It defines a color scheme and associates a name with that color scheme so that the scheme can be referred to by that name. Every color scheme that is used must be defined with PtDefCS before it can be used. For example, given the color scheme below:

GraphicsColor 0 and Background = Cyan
GraphicsColor 1 = Red
GraphicsColor 2 = White
GraphicsColor 3 = blacK

PtEchoed Background = Cyan
PtEchoed Foreground = White
PtNormal Background = Cyan
PtNormal Foreground = Red
PtLoud Background = Red
PtLoud Foreground = Cyan
PtQuiet Background = Cyan
PtQuiet Foreground = blacK

To define it with the name PtCS1, call PtDefCS as follows:
PtDefCS(PtCS1, 'CRWKCWCRRCCK');

The format for the string passed to PtDefCS can be any format the user desires. Any extraneous characters in the string (i.e., anything but K, B, G, C, R, M, Y or W) are removed, and whatever remains is used to determine what the color scheme is. Thus, extra characters may be added to a color scheme string to make it easier to read. For example, a more readable form of the call to PtDefCS given above would be:
PtDefCS(PtCS1, 'Cyan Red White blacK CW CR RC CK');

The Special Case of PtCSSystem

PtCSSystem is a special color scheme name because, as its name suggests, it represents the color scheme for the system message port. It is also the color scheme from which Ports picks up the information for the screen background color, the global palette, and the default color scheme. Thus, the screen background color is the same color as the one specified for PtCSSystem. The four drawing colors for PtCSSystem also become the global palette. In other words,

PtCSSystem's color scheme = default color scheme for Ports
GraphicsColor[0] = PaletteColor[0] = Screen background color
GraphicsColor[1] = PaletteColor[1]
GraphicsColor[2] = PaletteColor[2]
GraphicsColor[3] = PaletteColor[3]

Because the palette is defined by PtCSSystem, not only must there be a color scheme defined with the name PtCSSystem, but PTCSSYSTEM MUST BE DEFINED BEFORE ANY OTHER COLOR SCHEMES ARE DEFINED. Since PtDefCS needs the palette to define a color scheme, the information for the palette must already be available. Any colors scheme that are defined before PtCSSystem is defined will be set to a color scheme other than what is passed to PtDefCS.

IT IS IMPERATIVE the PtCSSystem color scheme be defined before PtInit is called and before defining any other color scheme! Since Ports gets the information for the screen background color, the global palette, and the system message port's color scheme from the color scheme named by PtCSSystem, it needs to know where to get this information. If Ports cannot find this information, the results will be bizarre and unpredicatable!

NOTE ON THE USE OF PTDEFINE WITH COLOR:

Writing a character in color on the edge of a port tends to make the character "bleed" into the screen background color because there is no border between the character and the screen background. This problem doesn't appear in black and white Ports because the screen background is the same color as the background of every port. With color Ports, however, it may be necessary to put a border around a port so that text written in it will not be on the edge of the port.

One way to do this is to define a port so that its boundaries are not on text boundaries. The extra area outside the text boundaries makes a border around the text. For example, if a port that needs a border around it is originally defined like this:

PtDefine(MessagePort, 5, 10, 70, 6, 1, []);

To put a border around it, define it like this:

PtDefine(MessagePort, 4.5, 9.5, 71, 7, 1, []);

This puts a border one-half character wide and high all the way around MessagePort. The border can be made anywhere from 0.1 characters to 0.9 characters wide and high, so it just depends on how wide you want to make your border.

If you have two ports which are next to each other and you want a border around each, make both borders no more than 0.5 characters wide so you will not lose more than one character between the borders. In fact, a 0.5 character border is rather nice.

EXAMPLE:

```
PtDefCS(PtCSSystem, 'CRWWKCWCRRCCK');
  { We must define this before any other color schemes }
PtDefCS(PtCS1, 'RCWKCKRCRWKC');
PtInit;   { PtInit is called after color schemes are set }
PtDefine(CyanPort, 5, 5, 70, 7, 1, []);
  { This ports use the default color scheme, PtCSSystem }
PtSetCrScheme(PtCS1);  { Set the color scheme to PtCS1 }
PtDefine(RedPort, 10, 16, 60, 5, 1, [Centered]);  { Uses PtCS1 }
PtSelect(CyanPort);  { Colors CyanPort's background to cyan }
...  { Statements doing text or graphics in CyanPort }
GrSelect(RedPort);   { Colors RedPort's backtground to red }
...  { Statements doing graphics in RedPort }
PtDisappear(RedPort);  { RedPort is colored to the screen color }
PtErase(CyanPort);   { CyanPort is erased to cyan }
TxpWrite('Now let's write in RedPort!', []);
  { RedPort is colored to red before the TxpWrite is done }
PtSelect(CyanPort);
  { CyanPort is already filled with cyan - no need to fill it }
TxpWrite('And let's write something in CyanPort', []);
PtErase(RedPort);   { Erase RedPort to red }
PtDisappear(CyanPort);  { Erase CyanPort to the screen color }
PtDispose(RedPort);  { RedPort is colored to screen color }
PtDispose(CyanPort); { CyanPort is not colored to screen color }
```

3.3 Input

OVERVIEW

All Ports unit routines which accept input from the user can enforce limits on the amount of time the user has to respond. When such limits are imposed, one says that the input operation is "timed." When such limits are exceeded, one says that the input operation has "timed out."

TxpAwaitUser, on the other hand, is timed (or not timed) based on data stored external to the program and read by PtInit as part of the initialization of the Ports unit. Thus, the programmer using this unit is neither responsible for nor in control of the timing that applies during TxpAwaitUser. What happens when this routine "times out" is described below in detail.

```
PROCEDURE TxpRead(VAR InString: TxpLongString; MaxLength: INTEGER;
                  Options: TxpOptSet);
```

TxpRead reads into InString from the CAT according to the set Options.
It first colors the CAT to its background color if it is not already filled with that color.
It then DISCARDS any keystrokes that may have been inadvertently entered before it was called.
Next, it starts reading at the current text cursor position within the port.
The user is allowed to enter a string which can be no longer than MaxLength characters.
TxpRead echoes the user's input in the PtEchoed text color. It does not change the current text writing color of the CAT.
The read operation may be terminated in one of four ways:

1) the user presses the RETURN key,
2) the read "times out" (described below).
3) the user presses the ESCKey requesting Extra Services Control and then selects an option which terminates the read,
4) MaxLength<=0 and the user types any "printable" character ("printable" is described below).

Note that the read operation is NOT automatically terminated simply because the user's response has reached the maximum length specified by MaxLength. Once the user's response reaches MaxLength characters, additional input will be ignored except for a tone that TxpRead generates. Thus, MaxLength should be generously large.

Printable characters are those in the ASCII range " " (ASCII 32) through "~" (ASCII 126) plus, on some systems but not all, those in the range ASCII 150 through ASCII 254. Characters beyond ASCII 127 are said to be in the "alternate character set." Under NO circumstances are characters in the range ASCII 128 through ASCII 149 considered "printable." According to international standards, that range is reserved for "control codes."

TxpRead generates a tone and ignores any character other than the

printable characters and the following 3 special keys:
RETURN -- signal end of user response;
ESCKey -- request Extra Service Control;
RubOut -- rub out/erase/delete previous printable character
(TxpRead generates a tone if there is no character to be deleted).

To allow the user to enter unprintable characters, a program should use TxpGetKey, the single key read routine.

When MaxLength > 0 and MaxLength printable characters have been entered, TxpRead responds to any further keystrokes (except the 3 special keys discussed above) by making a noise and ignoring the keystroke.

The text cursor is automatically displayed in the CAT at the start of a TxpRead operation in the PtEchoed text color, even if its previous status was "off." At the end of the TxpRead, it will be turned "off" if that was its status before the TxpRead.

The largest allowable value for MaxLength is 255 (a limit determined by UCSD Pascal). If MaxLength exceeds this value, string overflow errors may occur which are not immediately detected by the operating system and which thus may cause serious problems at a later (unrelated) portion of the program. In other words: Do NOT specify MaxLength>255!

WARNING:
It is EXTREMELY important that the programmer declare the string variable associated with InString to be of sufficient size to hold MaxLength characters.
Unpredictable and bizarre errors may result otherwise!
In most cases, both the string variable and MaxLength should be large enough so that only the most prolix user will be aware of any limitation on the length of his/her response.

TIMING AND TxpRead

TxpRead can enforce certain limits on the time allowed for user input during an invocation of this procedure. When these limits have been exceeded, the read has "timed out" (that is, TxpRead terminates, returning to the calling program control and whatever input the user has already entered).

These limits are best described in terms of two variables:
TxpReadTime and TxpStrokeTime.
These two variables are internal to the Ports unit.
They may be set via the procedure TxpSetTime.
Their current values may be queried via the procedure TxpGetTime. [3.8]

TxpRead will "time out" ONLY if BOTH of the following conditions are met:
a) the time since TxpRead began executing exceeds TxpReadTime [3.8] seconds AND
b) no key has been pressed within the last TxpStrokeTime seconds.
Thus, even if the user takes more than TxpReadTime seconds to complete a response, the read will NOT "time out" until there is no more input for TxpStrokeTime seconds.

In addition, if TxpStrokeTime > 0,
once the user has started to respond (pressed a key),
whenever there is no additional input for a period of TxpStrokeTime,
a reminder to press "Return" at the end of the response
will appear in the Ports unit message port.
This reminder remains displayed and is rewritten every TxpStrokeTime
seconds until the read terminates or the user presses the ESCkey.

TxpRead resets the CAT's I/O status indicator as follows:
0: Normal termination of read;
-1: After all input has been entered, unable to move to a new
line as requested by the EndLine [4] option;
-5: Read "timed out";
-6: Read terminated by user's Extra Service Control request.
The value of the I/O status indicator is queried via TxpStatus.

```
PROCEDURE TxpGetKey(VAR Key:CHAR);
```

TxpGetKey reads a single character from the keyboard, returning its value via the variable parameter Key.
It makes no noises, does no echoing, and does no filtering
EXCEPT for the transformations described below.
No textport options apply to TxpGetKey.
Unless TxpReadTime is 0 (see below), TxpGetKey starts by DISCARDING any keystrokes entered BEFORE it was called.
TxpGetKey is timed similarly to TxpRead with two exceptions:
1) The user has TxpReadTime seconds to respond;
the value of TxpStrokeTime is irrelevant.
2) When TxpReadTime = 0:
TxpGetKey does NOT discard keystrokes entered before it was called.
Instead, it returns the first key pressed since the previous read operation.
If no key was pressed since the previous read operation,
the port's I/O status is set to -5 (read timed out),
Key is set to CHR(0).

When the ESCkey is pressed, TxpGetKey responds as follows:
If the BOOLEAN variable ESCOk [2] in the Ports unit interface is TRUE (this is the normal case), the keystroke is taken as a request for Extra Service Control [3.9], that request is honored and then TxpGetKey resumes its wait for a key to be pressed. Thus, the ESCkey is handled by the Ports unit and not returned to the calling program; this is the usual situation.
If ESCOk is FALSE, TxpGetKey assigns its variable parameter Key the value CHR(ESC) (where ESC is a constant in the Ports unit interface [2]), sets the port's I/O status to -6 and then returns to the calling program. (An I/O status of -6 means the user selected a coder defined Extra Service Control option; when ESCOk is FALSE, ANY ESC request is considered a coder defined one.)

In addition to the special handling it gives ESCkey, TxpGetKey transforms certain function keys into special control codes:

Function Key	Control Code
Left Arrow	CHR(PtLeftArrow)
Right Arrow	CHR(PtRightArrow)
Up Arrow	CHR(PtUpArrow)
Down Arrow	CHR(PtDownArrow)

The INTEGER constants PtLeftArrow, PtRightArrow, PtUpArrow, and PtDownArrow are defined in the Ports unit interface [2].

WARNING:
Using TxpGetKey to get any character except the "printable" ones (as defined in our discussion of TxpRead) and those special keys just discussed SHOULD BE AVOIDED! It can EASILY LEAD TO PROBLEMS maintaining, enhancing and transporting to other systems programs which use these non-standard keys!

TxpGetKey resets the CAT's I/O status indicator as follows:
0: no error.
-5: TxpGetKey "timed out." That is,
TxpReadTime > 0 and TxpReadTime seconds elapsed without any input from the user.
TxpReadTime = 0 and no key pressed since previous read operation.
-6: TxpGetKey terminated by user's Extra Service Control request. (When ESCOk is FALSE, ANY ESC request is considered as terminating the routine.)
The value of the I/O status indicator is queried via TxpStatus.

EXAMPLE of using TxpGetKey to read arrow key:

```
REPEAT
  TxpWrite('Please press the left arrow key.', []);
  TxpGetKey(UserChar);
  IF TxpStatus = -5 THEN TxpWrite('You didn't press a key', [])
  ELSE { Some key was pressed. }
    IF UserChar = CHR(PtLeftArrow) THEN TxpWrite('That's it', [])
    ELSE TxpWrite('That was not the left arrow key',[]);
UNTIL ORD(UserChar)=PtLeftArrow;
```

EXAMPLE of using TxpGetKey to read key pressed BEFORE TxpGetKey is called:

```
TxpGetTime(Read, Stroke);
  { Save TxpReadTime and TxpStrokeTime values }
TxpSetTime(0, 0);
  { Then set them so TxpGetKey doesn't wait for a key to be
    pressed, but instead reports keys typed BEFORE it is called }
TxpWrite('To continue, please press the space bar. ', []);
REPEAT
  ShowPrettyPicture;
    { While we are doing this, user may press a key }
  REPEAT
```

```
    TxpGetKey(UserChar);  { check on any key ALREADY pressed }
        KeyPressed:= (TxpStatus=0);
        IF KeyPressed THEN { check what key was pressed }
          IF UserChar <> ' ' THEN TxpTone(0, 1);  { beep at wrong key }
      UNTIL NOT KeyPressed;
    UNTIL UserChar=' ';
    { Restore previous values of TxpReadTime and TxpStrokeTime }
       TxpSetTime(Read, Stroke);
```

PROCEDURE TxpAwaitUser;

TxpAwaitUser prints a message in the system message port asking the user to press the space bar when she/he is ready to continue. When the user presses the space bar, TxpAwaitUser terminates and the program goes on. During TxpAwaitUser, the user can press the ESCkey to request Extra Services Control. If the user selects the "Go on" or a programmer defined option from the ESC menu, TxpAwaitUser will terminate, reporting which, if any, programmer defined option was selected. Keys other than the space bar and the ESCkey are acknowledged by a noise but do not end the routine.

TxpAwaitUser does not "time out" in the way that TxpRead and TxpGetKey do. Instead, TxpAwaitUser keeps track of the time since it last detected any keystroke. When this time exceeds a predetermined limit, the routine asks the user to press any key just to show someone is still at the keyboard. If there is no input from the keyboard after two requests that the user press ANY key and a 20 second wait after each request, TxpAwaitUser TERMINATES THE ENTIRE PROGRAM!

This no-activity-for-X-minutes-and-TxpAwaitUser-stops-the-program feature and the value of X depends on data read from disk as part of the Ports system initialization performed by PtInit. This data may say that this feature should be disabled, that the program should wait for the user forever (or until the power fails). If PtInit cannot find the data, this feature is active.

TxpAwaitUser resets the CAT's I/O status [5] as follows:
0: no error.
-6: TxpAwaitUser terminated by user's Extra Service Control request.
The value of the I/O status indicator is queried via TxpStatus.

PROCEDURE GrCursor(VAR X,Y: REAL);

GrCursor allows the user to specify a point or position in the window of the CAG by using a graphics cursor (described below) as follows:

The initial position of the graphic cursor is determined either by the CGP or the coordinate (X, Y) depending on the value of the global boolean GrRubberbanding. If GrRubberbanding is TRUE, the graphic cursor will initially be located at (X, Y). If FALSE, it will be located at the CGP. However, if the point at which it should be initially displayed is not within the graphport, the graphic cursor

will be displayed in the center of the graphport.
The user moves the cursor by pressing the left, right, up and down arrow keys. The amount of movement per keystroke can be set using the GrStepSize routine. The default motion is 1 pixel per keystroke.
The user signals selection of a point by pressing the RETURN key; the "real world" coordinates of the selected point are returned to the calling program via the variable parameters X and Y. The CGP is NOT CHANGED!
The user can request Extra Service Control by pressing the ESCKey.
The time allowed for user input via GrCursor is handled in the same fashion as it is with TxpRead. That is, GrCursor enforces the time limits specified by TxpReadTime and TxpStrokeTime as they are set via TxpSetTime. If GrCursor "times out," the CAT's I/O status is set to -5 and the values of X and Y indicate where the graphics cursor was when GrCursor "timed out."
The graphics cursor is a set of small flashing cross-hairs in the current line-drawing color. If the Ports unit BOOLEAN variable GrRubberBanding is TRUE, the center of these cross-hairs is connected by a line (the "rubber band") to the CGP. The line drawing style and color of the rubber band is that of the CAG.
When GrCursor terminates, all components of the graphics cursor are removed from the screen.

NOTES:
GrCursor does not affect the CGP in any way. The only relation between the two is that the CGP is used to determine the initial position of the graphics cursor and thus is also one end of the rubber band if a rubber band cursor is used.
GrCursor does not allow any part of the cursor to be moved outside the CGP. Thus there is a strip of pixels at the perimeter of the port into which the cursor will not go. The width of this strip depends on the hardware being used and the cursor step sizes.
If the CAG's window has not been defined or the port is so small that the cursor will not fit into it, GrCursor returns immediately, without doing anything.
The graphic and text cursors are independent.

GrCursor resets the CAT's I/O status indicator as follows:
0: no error.
-5: GrCursor "timed out."
-6: GrCursor terminated by user's Extra Service Control request. (When ESCOk is FALSE, ANY ESC request is considered as terminating the routine.)
The value of the I/O status indicator is queried via TxpStatus.

EXAMPLE:

```
TxpWrite('Please enter a triangle.', []);
TxpWrite('  Start with one point.', []);
   GrCursor(X1,Y1);
   MoveTo(X1,Y1);  { Since GrCursor doesn't move CGP, we must }
   TxpWrite('  OK, now another.', []);
```

```
      GrCursor(X2,Y2);
      GrLineTo(X2,Y2);  { Draw first side }
   TxpWrite('  Good.  Now the third.', []);
      GrCursor(X3,Y3);
      GrLineTo(X3,Y3);  { Draw second side }
   TxpWrite('  Fine.  Now connect back to the first point.', []);
      OnTarget:= FALSE;
      REPEAT
         GrCursor(X,Y);
         IF ABS(X-X1)+ABS(Y-Y1) <= Tolerance THEN OnTarget:= TRUE
         ELSE TxpWrite('  Not close enough.  Try again.', [])
      UNTIL OnTarget;
      LineTo(X1,Y1);  { Draw third side }
```

```
PROCEDURE GrStepSize(XStep, YStep: REAL);
```

GrStepSize sets the size of each movement made by the graphics cursor when it is in the port which is the CAG. Once set for a particular port, these values are "remembered" by the port even when it is no longer the CAG. Thus, once these values have been set, there is NO NEED to call GrStepSize each time the port is selected. However, the step sizes are reset to their default value (see below) whenever GrWindow is applied to the port.

XStep's absolute value specifies how far the graphics cursor moves horizontally in response to the left or right arrow keys; ABS(YStep) specifies the amount of vertical motion in response to the up or down arrow keys. The units in which XStep and YStep are measured depend on the signs of these numbers:

XStep	YStep	Meaning
0	0	Default setting: move one pixel in each direction in response to corresponding arrow key
>=0	>=0	Units of both XStep and YStep are in "real world" coordinates as defined by GrWindow
<=0	<=0	Units of both XStep and YStep are in pixels.
>0	<0	Units of XStep are in "real world" coordinates, YStep is adjusted so that motion is isotropic.
<0	>0	Units of YStep are in "real world" coordinates, XStep is adjusted so that motion is isotropic.

NOTES:

Here, "isotropic" means that each horizontal step and each vertical step cover (approximately) the same number of cm. on the screen.

If one, but not both, of XStep and YStep is 0, the graphics cursor can move ONLY horizontally or ONLY vertically!

Small non-zero values of XStep or YStep are adjusted to assure that some motion is possible in the corresponding direction.

If you want NO MOTION, make sure that the value of XStep or YStep is EXACTLY 0.

Most of these cases can be summarized as follows:
(0,0) always means "default;"
positive means "real world;"
0 means no movement in that direction;
negative means units are pixels;
with differing signs, the positive one takes precedence and the other direction is adjusted so that motion in each direction is the same for each keystroke in that direction.

3.4 Output

```
PROCEDURE TxpWrite(OutString: TxpLongString; Options: TxpOptSet);
```

TxpWrite first colors the CAT to its background color if it is not already filled with that color. It then writes OutString in the CAT according to the textport options in Options. OutString is written in the current text writing color. If it is physically possible, TxpWrite will break a string which cannot fit on a single line at a space ("smart word wrap"). To force a portion of a string with embedded spaces to appear on a single line, the programmer may use "sticky spaces." A sticky space is denoted by the reverse accent, "`" (ASCII character 96). TxpWrite displays this character as though it were a space, " " (ASCII character 32) but does NOT use the sticky space as marking an appropriate place to break a string which doesn't fit on a single line. Thus, the effect of using a sticky space is to bind separate words together so that they will not be broken across two lines if this can be avoided.

TxpWrite resets the CAT's I/O status indicator as follows:
- 0: Normal termination of write;
- -1: After the entire string has been written, unable to move to a newline as requested via the EndLine [4] option;
- -6: User selected a coder defined option during ESC request;
- N>0: Forced to terminate write before outputting the last N characters of OutString because of lack of space in port and because of setting of scrolling option.

The value of the I/O status indicator is queried via TxpStatus.

```
FUNCTION TxpCursor(NewSetting: BOOLEAN): BOOLEAN;
```

On some systems, it is possible to control whether or not a text cursor is displayed at the CTP in the CAT. On other systems, there is no way to control whether or not the text cursor is displayed. On all systems, however, a text cursor will ALWAYS be displayed as part of the TxpRead procedure. The cursor will appear in the current text color.

On systems which allow one to turn off and on the text cursor, it will be displayed ONLY during TxpAwaitUser, during the normal "Quit?" option in Extra Services Control, and (the principal case) when the program is accepting input from the user via the TxpRead procedure. On such systems, calling this function turns ON the display of the text cursor OUTSIDE of TxpRead (when NewSetting is TRUE) or OFF (when NewSetting is FALSE) and returns as its value the previous setting:
- FALSE for the (normal) "display only during TxpRead;"
- TRUE for the "display at all times."

This value can be used in a later call to TxpCursor to reset the display/not-display to its previous status; restoring values that you alter when you are done with them is good programming practice.

WARNING: Because TxpRead automatically handles displaying the text cursor as part of showing the user that the program is awaiting his/her response, IT IS RARELY NECESSARY TO USE THIS ROUTINE.

EXAMPLE:

```
TextCursor:= TxpCursor(TRUE);
... { statements during which the text cursor is always displayed }
TextCursor:= TxpCursor(TextCursor);  { restore previous setting }
```

PROCEDURE TxpTone(Pitch, Duration: INTEGER);

TxpTone produces a tone whose pitch and duration are determined by the Pitch and Duration parameters respectively. If Pitch=0, the tone is the standard system "bell"; other pitches have not yet been implemented and are likely to be system dependent also. No tone is produced if Duration=0.

OVERVIEW OF GRAPHICS OUTPUT

Before a port is used for any graphics other than being framed (GrFrame) or erased (PtErase), the program must establish a "window" for it by calling GrWindow. Once that is done, until the port is discarded (via PtDispose), the system maintains a Current Graphic Point (CGP) for that port. The graphics positioning routines (GrMove and GrMoveTo) and the graphics output routines (GrArc, GrLine, and GrLineTo) all refer to and update the CGP. The graphics input routine (GrCursor) refers to the CGP but does not update it.

The type of line drawn by the graphics output routines described in this section is determined by the line "color", "mode" and "style specified for the CAG. The values for the line "color" are given by values for the line "mode" are given by the enumerated data type GrLnMode, and the values for the line "style," by GrLnStyle. The default values for line color, mode, and style produce a solid line in line color 1 on the port background color. The line color, mode and style for the CAG can be changed with GrSetLnColor, GrSetLnMode and GrSetLnStyle respectively. The new settings are associated with the port which is the CAG at the time these procedures are called and remain in effect for that port even when it is no longer the CAG, until changed explicitly.

PROCEDURE GrSetLnMode(NewMode:GrLnMode);

GrSetLnMode sets to NewMode the line mode of the port which is the CAG.
Subsequent drawing in this port is done in this mode.
The line mode may be changed as often as desired.
Only the port which is the CAG is affected and this effect will not be apparent until the next graphic output in that port.

PROCEDURE GrSetLnStyle(NewStyle: GrLnStyle);

GrSetLnStyle sets to NewStyle the line style of the port which is the CAG.
Subsequent drawing in this port is done in this style.

The line style may be changed as often as desired.
Only the port which is the CAG is affected and this effect will not be apparent until the next graphic output in that port.

```
PROCEDURE GrArc(XDistCenter,YDistCenter,Angle:REAL);
```

GrArc draws in the CAG a circular arc starting at the current graphic point (CGP). The center of the arc is (XDistCenter, YDistCenter) AWAY FROM the current point. The arc sweeps out Angle degrees; it is drawn counterclockwise for Angle > 0, clockwise for Angle < 0. The arc is drawn according to the port's line color, style and mode (as described above). Then the CGP point is shifted to the end of the arc, even if that point does not lie within the port's window.
NOTE: Since arcs are drawn as sequences of short line segments, dotted or dashed arcs may not look uniformly dotted or dashed.

```
PROCEDURE GrLineTo(NewX,NewY:REAL);
```

GrLineTo draws a line from the CGP of the CAG to the point (NewX, NewY) according to the port's line color, style and mode (as discussed above). It then updates the port's CGP to the new point, (NewX, NewY), even if that point does not lie within the port's window.

```
PROCEDURE GrLine(XDistance,YDistance:REAL);
```

GrLineTo draws a line from the CGP of the CAG to the point displaced from the CGP XDistance "real world" units in the X (horizontal) direction and YDistance "real world" units in the Y (vertical) direction, according to the port's line color, style and mode (as discussed above). It then updates the port's CGP to this new point, even if that point not lie within the port's window.

GrLine(0,0) just draws a dot, without changing the CGP.

```
PROCEDURE GrFrame;
```

GrFrame draws a frame at the perimeter of the CAG. The frame is drawn in the port's current line color, mode and style. The CGP is not affected. The port need not have a window specified.

```
PROCEDURE GrBox(WhatPort: Port;
                StartCol, StartRow, Width, Height: REAL;
                FlashCount: INTEGER);
```

Puts a box on the screen. If WhatPort is NIL, StartCol and StartRow are interpreted as screen text coordinates, and the color scheme and line drawing mode are taken from the CAT. If WhatPort is non-NIL, StartCol and StartRow are interpreted as text coordinates of WhatPort, and the color scheme and line drawing mode are taken from WhatPort. The box

is always drawn in XOR mode. FlashCount determines how many times the box is drawn. If FlashCount is even, no box is left on the screen; if FlashCount is odd, a box is left on the screen.

```
FUNCTION GrClipping(NewSetting: BOOLEAN): BOOLEAN;
```

Normally, before any Ports unit routine does any drawing, it automatically "clips" any portion of the drawing that does not lie within the CAG's window. Checking to see what parts of the drawing may lie outside the window does slow the drawing down a bit and may not always be necessary. In those situations where the programmer is ABSOLUTELY SURE that all parts of the drawing the program is about to display will lie entirely within the CAG's window, he/she can use GrClipping to suppress clipping and allow the drawing to go faster.

NewSetting indicates whether clipping is to be performed (TRUE) or suppressed (FALSE). The value returned by GrClipping reports what the previous status of clipping was. This value can be used in a later call to GrClipping to restore clipping to its previous status; restoring values that you alter when you are done with them is good programming practice.

Because drawing without clipping can be dangerous (it can cause a program to misbehave in unpredictable ways), clipping is suppressed ONLY UNTIL the next GrSelect statement or until a GrClipping(TRUE) is executed. That is, if you want to suppress clipping, you must do so after you have selected the port where you want it suppressed and you must suppress it EACH time you select the port.

WARNING:
The direst sort of disaster can befall a program which turns clipping OFF (GrClipping(FALSE)) and then draws something that does not lie entirely within the CAG.
NEVER turn clipping off UNLESS you are ABSOLUTELY SURE that ALL DRAWING in the CAG will lie ENTIRELY INSIDE that port given the windowing associated with it!

EXAMPLE:

```
Clip:= GrClipping(FALSE);
... { statements we are ABSOLUTELY SURE NEVER DRAW OUTSIDE the CAG }
Clip:= GrClipping(Clip);  { reset clipping to prior value }
...
Clip:= GrClipping(FALSE);
GrSelect(APort);  { clipping is automatically turned back on }
... { these statements may not always draw entirely within APort }
Clip:= GrClipping(FALSE);
... { statements we are ABSOLUTELY SURE NEVER DRAW OUTSIDE APort }
Clip:= GrClipping(TRUE);
... { these statements may not always draw entirely within APort }
```

3.5 Query

```
FUNCTION TxpStatus: INTEGER;
```

The value returned by TxpStatus is the value of the CAT's I/O status indicator. Section [5] summarizes these values, their meanings and which routines set them.

EXAMPLE:

```
TxpRead(Answer,MaxSizeAnswer,[]);
IF TxpStatus = -5 THEN BEGIN  { Read timed out }
   IF LENGTH(Answer) > 0 THEN
      TxpWrite('You forgot to press the RETURN key', [])
   ELSE BEGIN
      TxpWrite('Please don''t fall asleep on us!', []);
      FOR I:= 1 TO 5 DO TxpTone(0, 0);
      TxpAwaitUser;  { Wait until user is ready or TxpAwaitUser decides
                       the user has left and the program should end }
      END;
   END;
```

```
FUNCTION TxpLineLast: BOOLEAN;
```

TxpLineLast answers the question "Is this line the LAST one possible in the current port?" It returns the value TRUE if the current line of the currently active textport is the last line, i.e., the current row is the last one in the space allocated to the port on the screen and the scroll option in effect at the moment is DemandScroll or NoScroll [4].

```
FUNCTION TxpWhatPort: Port;
```

TxpWhatPort returns, as its value, the currently active textport.

EXAMPLE:

```
IF TxpLineLast THEN BEGIN
   SavedPort:= TxpWhatPort;
   TxpSelect(ExtraPort);
   TxpWrite(ImportantMessage, []);
   TxpSelect(SavedPort);
   END
ELSE
   TxpWrite(ImportantMessage, []);
```

```
FUNCTION TxpTicSize(NewValue: INTEGER): INTEGER;
```

TxpTicSize returns the value of the Ports unit's internal timing

variable, TxpTicToc, at the time the function was invoked. If NewValue>=0, then this function resets TxpTicToc to NewValue.

TxpTicToc controls the overall speed of presentation of text. It is always non-negative; the larger it is, the slower is the speed. It can affect how fast TxpWrite writes and how long TxpPause pauses. By asking to go faster or slower, the user decreases or increases TxpTicToc thereby shortening or lengthening variable pauses as well as speeding up or slowing down the speed at which TxpWrite displays text when the Slow option is in effect.

"Normal" writing speed corresponds to TxpTicToc = 10.

EXAMPLE of varying and then restoring writing speed:

```
Original:= TxpTicSize(10);  { 10 is "normal" writing speed }
   ...  { Write text at "normal" speed }
Slowing:= TxpTicSize(Slowing);
Slowing:= TxpTicSize(Original*2);
   ...  { Write text with approximately TWICE the slowing
          that the user chose originally }
Slowing:= TxpTicSize(Original);  { Restore original speed }
```

```
FUNCTION GrWhatPort: Port;
```

GrWhatPort returns, as its value, the currently active graphport.

EXAMPLE:

```
PROCEDURE FramePhrase(Column, Row, LenPhrase: INTEGER);
{ Draw a box one text column high and LenPhrase columns wide
  whose upper left corner is at (Column, Row). }
VAR SavePort, Framer: Port;
BEGIN
   SavePort:= GrWhatPort;
   PtDefine(Framer, Column, Row, LenPhrase, 1, []);
   GrSelect(Framer);
   GrFrame;  { We don't NEED to window the port before framing it }
   GrSelect(SavePort);  { Restore CAG to its saved value }
   PtDispose(Framer);  { Give back memory that Framer required }
END; { FramePhrase }
```

```
FUNCTION PtWhatPort: Port;
```

PtWhatPort returns the CAP, unless there is none (because the CAT and CAG are not the same) in which case it returns the value NIL.

3.6 Positioning

```
PROCEDURE TxpGoTo(X,Y: INTEGER);
```

TxpGoTo updates the CTP to be column X and line Y of the CAT if that position is within the textport; otherwise, the CTP is not changed.

TxpGoTo resets the textport's I/O status [5] indicator as follows:
0: Goto successful.
-3: Goto unsuccessful; specified column not within the textport.
-4: Goto unsuccessful; specified column BUT NOT specified line currently within the textport.

```
PROCEDURE TxpWhereAmI(VAR X,Y: INTEGER);
```

TxpWhereAmI sets X and Y to the column and line components respectively of the CAT's CTP. The top left position of a textport is location (0, 0) if no scrolling has taken place. The "line" is the sum of the number of rows that have already been scrolled out the top of the port plus the "physical row" (actual current row position within the port).

```
PROCEDURE TxpNextLine;
```

The CTP of the CAT will be moved to start of the next line of the port if possible, scrolling if necessary. (If this port has the DoubleSpacing [4] option set, "next line" means two lines down.) The only condition under which this routine will not move to the next line is if scrolling is required and the scrolling option in effect is either DemandScroll or NoScroll [4].

TxpNextLine resets the port's I/O status to indicate success (TxpStatus = 0) or failure (TxpStatus = -1).

```
PROCEDURE GrMoveTo(NewX,NewY:REAL);
```

GrMoveTo moves the CAG's CGP to the point (NewX, NewY). This new point need not lie within the window. Moving draws nothing.

```
PROCEDURE GrMove(XDistance,YDistance:REAL);
```

GrMove moves the CAG's CGP XDistance "real world" units in the X (horizontal) direction and YDistance "real world" units in the Y (vertical) direction. Moving draws nothing.

```
PROCEDURE GrWhereAmI(VAR NewX,NewY:REAL);
```

GrWhereAmI returns the position of the CAG's CGP in "real world" coordinates.

3.7 Clearing

PROCEDURE PtErase(PortName: Port);

Clears the text and graphics from the PortName by filling the port with its background color. The CTP is reset to the first line, first column. The CGP is not changed.

PROCEDURE PtDisappear(PortName: Port);

Clears the text and graphics from PortName by filling it with the screen background color if it has not already been done (either with PtEraseAll or PtDisappear). This has the effect of "disappearing" the port. The CTP is reset to the first line, first column. The CGP is not changed.

PROCEDURE PtEraseAll;

Erases the entire screen to the screen background color, erasing all text and graphics from all ports. The CTP's in each defined port are reset to the first line, first column; the CGP's are not changed.

PROCEDURE TxpScroll(Size: INTEGER);

TxpScroll scrolls the CAT according to the type of scrolling specified for it. Size is the number of lines to be scrolled; 0 means to use the default size. If Size exceeds the CAT's height, it is set equal to the CAT's height. The scrolling will not be done if the NoScroll [4] option is in effect. If the DemandScroll [4] option is set in a port, this routine must be called any time scrolling is desired. The lines scrolled in are in the CAT's background color.

PROCEDURE TxpClrPt;

Clear the CAT from the CTP through the end of the textport to the port background color. The CTP is not changed.

PROCEDURE TxpClrLn;

Clear the CAT from the CTP through the end of the current line to the port background color. The CTP is not changed.

3.8 Timing

PROCEDURE TxpSetTime(ReadTime,StrokeTime: INTEGER);

This procedure sets TxpReadTime and TxpStrokeTime to ReadTime and StrokeTime seconds respectively:
TxpStrokeTime is set to ABS(StokeTime).
TxpReadTime is then set so that TxpReadTime >= TxpStrokeTime.
TxpReadTime and TxpStrokeTime are variables that are internal to the Ports unit. Various Ports unit routines (TxpRead, TxpGetKey and GrCursor) use them in timing input from the user.

The values that TxpReadTime assumes have varying effects on the way input routines do their timing. These values fall into three classes:
0: meaningful only to TxpGetKey [3.3]. Signifies that the read is to be done asynchronously [8] (return the last key pressed since the last read).
1 <= TxpReadTime <= 3000: "Normal" values for TxpReadTime. The number of seconds to wait before "timing out" [8].
3001 <= TxpReadTime: all input routines assume an "infinite" read time. No "time outs" will occur. (NOTE: the programmer should never have to set TxpReadTime to an "infinite" value. If the suppression of "time outs" is desired, a better way to accomplish it is through TxpDebug [7].)

PROCEDURE TxpGetTime(VAR ReadTime, StrokeTime: INTEGER);

TxpGetTime returns the current values of TxpReadTime and TxpStrokeTime.

PROCEDURE TimerSet(VAR Timer: TimerType);

This procedure sets the TimerType variable Timer to the current value of the computer's internal clock. It is used in conjunction with later invocations of the TimerRead routine to measure elapsed time. This procedure is hardware dependent: if the host lacks an internal clock, TimerSet will set Timer to 0.0.

FUNCTION TimerRead(Start: TimerType): INTEGER;

This function returns the number of seconds since Start was set by TimerSet. This function is hardware dependent: if the host lacks a clock, this routine will not alter the value of Start.

PROCEDURE TxpPause(HowLong: INTEGER);

This procedure causes the program to pause (do nothing, wait). The length of the pause is given by HowLong. A TxpPause(1) causes the program to wait about the writing time of two words, as reflected by

the current writing speed. That is, the length of the pause depends on the current writing speed which the user can change to suit his/her taste. Because of the measure of user control this type of pause allows, it is preferred over the absolute delay dictated by TxpDelay (discussed below). The MINIMUM length of a pause is about 1/2 second.

```
PROCEDURE TxpDelay(HowLong: INTEGER);
```

This procedure causes the program to pause (do nothing, wait). The length of the pause is HowLong tenths of seconds. There is no minimum length for this pause, so a 0 parameter to TxpDelay is legal.

NOTE:
The actual clock time that elapses from the time a program starts executing the statement "TxpDelay(HowLong)" until it completes the execution may vary from HowLong/10 seconds by as much as 5% or 0.05 seconds, whichever is larger. The reason for this is the overhead inherent in any procedure call combined with limitations in the accuracy of the hardware clocks being used. Thus, while TxpDelay(25) will take reasonably close to 2.5 seconds to execute, the programmer must expect that execution of the following loop will take considerably more than 2.5 seconds:

```
FOR I:= 1 TO 25 DO TxpDelay(1);
```

3.9 Extra Services Control

The Ports unit allows the user to interrupt normal program flow by pressing the Extra Services Control key which we denote ESCkey. On most systems, the key used as the ESCkey is label "ESC" which is usually located in the upper left corner of the keyboard and called the "Escape" key. We use the work "ESCkey" to distinguish this key from the the phrase "Extra Service Control" and the standard abbreviation for ASCII code CHR(27).

A word of warning is in order: This section describes a reasonably complex and flexible set of capabilities, most of which are used only rarely. The capabilities that are most frequently useful are described later in this section under the heading "Special Debugging Features."

A program's response to a user's pressing the ESCkey depends on the current value of two internal system variables, ESCPrompt and ESCSet, and on what operation (routine) is being interrupted. ESCPrompt is a string variable. ESCSet is a variable of type Alphabet. The only access a program has to these variables is via the TxpESCSet procedure (described below).

The following routines can be interrupted by a user's request for Extra Service Control:

TxpRead, TxpGetKey, TxpAwaitUser, GrCursor,
TxpWrite, TxpScroll, TxpPause, TxpDelay.

When the user presses the ESCkey, normal execution of these routines is interrupted and the following message is displayed in the system message port in its PtLoud text mode:

<ESCPrompt> "Writing speed? Quit? Go on? ".

(We use the notation "<ESCPrompt>" to indicate the value of ESCPrompt.) Then the text cursor is placed after this message and the user makes a request by typing in a single letter.

Sections of this prompt may be omitted and different actions taken depending on the values of ESCPrompt and ESCSet. Programmer-defined options may be specified by adding letters to ESCSet and changing the value of ESCPrompt. For example,

To add the option "Help" to the default set of actions,
the programmer should use the TxpESCSet procedure to
1) add the letter 'H' to ESCSet and
2) change ESCPrompt to "Help? ".

A programmer-defined option that conflicts with a default option has a higher precedence. Thus, adding the letters 'W', 'Q', or 'G' to ESCSet has the effect of omitting their respective default prompts from the end of the message displayed and allowing/forcing the programmer to handle them directly as described below. For instance, if 'Q' is in ESCSet, the "Quit?" part of this message is omitted.

All user responses are converted to upper case before checking their value and it is in this form that they are reported by the TxpESCRequest function.

When 'W' for "Writing speed" is typed, the following choice is presented:

"Faster writing? Slower writing? Go on? "

This prompt is written in the system message port's PtLoud text mode, and a single letter selects an option:
"F" and "S" adjust the writing speed and the choice is redisplayed at the newly specified speed;
"G" returns control to the interrupted routine.
If 'W' is in ESCSet, the writing speed prompt is suppressed and the user is no longer able to adjust the writing speed as part of Extra Service Control.

When 'Q' for "Quit" is typed, the system asks (once again, in PtLoud mode):
"Shall we STOP (YES/NO)? "
and waits for the user's response, repeating the message if the user answers anything other than "YES", "NO", or simply pressing RETURN (this last case is considered a "NO"). If 'Q' is in ESCSet, the quit message is not displayed; instead, "Q" is considered as a programmer-defined option. Thus, if program logic requires that a "quit" option be handled by the program directly and not by the normal ESC mechanism, the programmer can set ESCSet to include 'Q' and ESCPrompt to show the user that a "Quit" option is available. The ESC routine will consider "Q" as selecting a programmer-defined option and return control to the calling program to handle the user's request to "quit" as it sees fit.

When the user enters a letter which is in ESCSet, this letter is saved in ESCChar (a variable of type CHAR internal to the unit), the CAT's I/O status is set to - 6, and control is returned to the interrupted Port unit routine. The action this routine then takes is given in a table below. Letters other than "W", "Q", and "G" which are not in ESCSet are discarded after the ESC routine makes a noise and redisplays its prompt. The programmer can access the value of ESCChar by using the TxpESCRequest function.

Routine	Action after return from ESC request
TxpRead, GrCursor	Terminates if a programmer-defined option is selected; otherwise, continues to normal completion.
TxpGetKey	Returns ASCII code CHR(0) if a programmer-defined option is selected; otherwise, continues to normal termination.
TxpAwaitUser	Terminates.
TxpWrite	Continues to normal termination.
TxpScroll	Continues to normal termination (i.e., scrolls).
TxpPause	Terminates.
TxpDelay	Terminates.

ESCChar is set to ' ' at the beginning of each ESC request. Thus, if two ESC requests are made with no checking between them, only the most recent selection is available for checking.

Generally, ESCPrompt should list the options only the program itself can provide to the user. Each option should start with a distinct letter (not "G" or "W") and ESCSet should contain these letters. 'Q' should be used only for "Quit" whether this option will be handled directly by the program or left to the Ports unit. Given the use of "F" and "S" to

set writing speed as described above, it is probably best to avoid these letters although TxpESCSet, the procedure used to set ESCPrompt and ESCSet, does not enforce this structure.

When ESCOk (a BOOLEAN variable in the unit's interface) is FALSE, ANY ESC request is considered as selecting a programmer-defined option, the CAT's I/O status is set to -6, and control is returned to the unit), the CAT's I/O status is set to -6, and control is returned to the interrupted Port unit routine which behaves as described above.

Special Debugging Features:

There are occasions, particularly when debugging a program, when it is convenient to find out more information about a program's operating environment.

On such occasions, press the "(" during the ESC routine. Three things will happen:

The Ports unit internal BOOLEAN variable TxpDebug will be set TRUE (your program can subsequently test this variable and when it is TRUE, output information that, while helpful in debugging, is not appropriate in a completed program);

The ESC routine will respond with the following message:

Current Ports version number/TxpMsgName/###### words

where TxpMsgName is the name of the last message displayed using the Keyed File message display facility (the name will be empty if this facility isn't being used or no messages have been displayed since the start of the program), and ###### is the number of words of memory remaining.

All ports will be frames in the dotted line style whenever they are erased or colored for the first time. This feature helps the programmer see where ports overlap and when they are colored.

Pressing any key now returns control to the calling routine.

When TxpDebug is TRUE, no timed routine in the Ports unit will "time out."

To set TxpDebug FALSE, press ")" during the ESC routine.

(TxpDebug is discussed more completely in section 8.)

Summary of internal (hidden) variables used by Extra Service Control (ESC):

```
ESCPrompt: STRING;      Variable part of ESC message;
ESCSet: Alphabet;       Responses to ESC message which will be
                        handled by the program using Ports unit;
ESCChar: CHAR;          Most recent user selection from ESCSet.
```

The following procedures access these internal variables:

PROCEDURE TxpESC;

TxpESC generates an Extra Services Request under program control. This means that this procedure, when called, will act exactly as if the user had pressed the ESCkey. The current value of ESCOk is not relevant.

```
PROCEDURE TxpESCSet(PromptMessage: STRING; Letters: Alphabet);
```

Txp ESCSet changes how the Ports unit responds when the user presses the ESCkey. ESCPrompt is set to PromptMessage and ESCSet is set to Letters. ESCChar is set to ' '. The role of these three internal variables is fully discussed above.

```
FUNCTION TxpESCRequest: CHAR;
```

TxpESCRequest returns ESCChar. In general, ESCChar contains the last option-selecting character typed by the user as part of an Extra Service Control request.

EXAMPLE:

```
TxpWrite('This ends our discussion of computer literacy.', []);
TxpESCSet('Begin again? Help? Quit?', ['B','H','Q']);
   { Extra Service Request prompt line will read
        "Begin again? Help? Quit? Writing Speed? Go on? "  }
TxpESC;
IF TxpStatus = -6 THEN
   CASE TxpESCRequest OF
      'B': TxpWrite('I refuse to begin that again!', []);
      'H': TxpWrite('I cannot help it any more than you can!', []);
      'Q': TxpWrite('It''s too late to quit.  Onward!', []);
      END
ELSE
   TxpWrite('O.K., we''ll go on (and on (and on (...)))', []);
```

3.10 System Messages

The Ports unit displays its own messages and directly interprets a user's responses to them in the following situations:

Remind user: Remind the user to press the return key after he/she is finished entering input. This occurs when a read is in the process of "timing out". See section 4.3 of this document for a description of the exact circumstances under which this occurs.

Wait for user: Wait for the user to press the space bar to indicate that he/she is ready to go on. A program creates this type of wait by calling the TxpAwaitUser routine; for details, see section 4.3 of this document.

Extra services: When the user presses the "extra service control" (ESC) key, Ports makes certain "extra services" available; see section 4.9 of this document for a description of these services and how a user requests them.

Chaining message: When ChainTo [3.11] is called, a message is displayed in the system message port asking the user to wait one moment.

In order for dialogs using the Ports unit to be able to work in languages other than English, the Ports unit itself must be able to display these messages and interpret user responses to them in more than one language. All these messages and user responses can be specified by strings (called "system strings" in this discussion) which are internal to the Port unit. Each system string is identified by a single character mnemonic tag:

Tag	Default (English) Value of System String	Situation
R	When you're done, press Return.	Remind user
1	To continue, please press the space bar.	Wait for user
2	Please press a key so we know you are there.	Wait for user
3	If you don't press a key, the program will stop.	Wait for user
G	Go on?	Extra services
W	Writing speed?	Extra services
F	Faster writing?	Extra services
S	Slower writing?	Extra services
Q	Quit?	Extra services
A	ABCDEFGHIJKLMNOPQRSTUVWXYZ	Extra services
?	Shall we STOP (YES/NO)?	Extra services
C	One moment, please.	Chain to

The role of each system string is described below.

Remind User Operation:
The system string with tag "R" for reminder is displayed. The Ports unit handling of the user's response to this message is language independent.

Wait For User Operation:
Essentially, the system strings with tags "1", "2" and "3" represent three levels of waiting after which the Ports unit assumes that there is no one at the keyboard.
Initially, Ports displays the system string with tag "1" telling the user to press the space bar when he/she is ready to go on.
After a substantial delay, it displays the system string with tag "2" and still later the system string with tag "3" trying to determine if there is anyone at the keyboard. Section 4.2 of this document provides full details about this.
The Ports unit handling of the user's response to this message is language independent.

Extra Services Operation:
The system strings with tags "G", "W", "F", "S", and "Q" are used as items in menus. The user selects an option from a menu by pressing a single key. Customarily this key is the first letter in the menu item. Single letter user responses within the extra service control routine are transformed according to the system string with tag "A" as described below.

The system string with tag "A" is called the "alphabet string." It specifies how single letter user responses within extra service control (ESC) requests should be transformed before they are analyzed and acted upon. When the ESC handling routine in Ports gets as a single letter response the Nth letter in the English alphabet, it substitutes for this response the Nth letter in the "alphabet string." This transform maps single letters which are appropriate responses for menu selections in language X (X might be Italian, French, Spanish, Hungarian, or even English) into the letter which is appropriate in an English menu. Thus, the alphabet string MUST BE EXACTLY 26 characters long.

For example, before using "Veloce? Lento? Continuare?" as the Italian form of "Faster writing? Slower writing? Go on?", Ports must be told to transform "V" into "F", "L" into "S", and "C" into "G". This is done by setting the alphabet string to 'ABGDE--HIJKSMNOPQR-TUFWXYZ'. In order to keep the letters "F", "G", and "S" which have no meaning in our use of Italian from being taken as appropriate responses from a user, the alphabet string specifies that they be transformed into "-". Another, more complete, example is given below.

The system string with tag "?" asks the user to confirm that he/she wants to quit the program. It must contain the two (positive and negative) responses the user is allowed to give specified in the following fixed form:

-- Upper case only;

-- No embedded blanks;
-- Positive response immediately preceded by "(";
-- Negative response immediately followed by ")";
-- Responses separated only by "/";
-- No other "(", "/", or ")" in the string.

For example, in Italian this system string might be
'Vuoi proprio interrompere il programma (SI/NO) ?'
The restrictions given above must be followed RIGOROUSLY!

Chaining Operation:
The system string with tag "C" for chaining is displayed.
No response is solicited from the user.

The Ports unit has two routines which allow the programmer access to the system strings:

PROCEDURE PtSetString (Tag: CHAR; AString: STRING);
PtSetString sets the system string identified by Tag to the value given by AString. If the value of Tag is not one of the characters specified above, PtSetString does nothing.
WARNING: There are RIGID constraints (described above) on the format of the system strings with tags 'A' and '?'. Using PtSetString with strings which do not conform to these constraints can give VERY UNPLEASANT results.

PROCEDURE PtGetString (Tag: CHAR; VAR AString: STRING);
PtGetString sets AString to the current value of the system string identified by Tag. If the value of Tag is not one of the characters specified above, PtGetString does nothing.

EXAMPLE: Setting Ports System Strings in Italian:

```
  { Set string used to remind user to press Return at end of input }
PtSetString('R', ' Quando hai finito, premi Return. ');
  { Set strings used in waiting for user }
PtSetString('1', ' Per continuare, premi la barra per gli spazi. ');
PtSetString('2', ' Premi un tasto per piacere, cosi sappiamo che ci sei.
PtSetString('3', ' Se non premi un tasto, il programma s''interrompe. ')
  { Set strings used in ESC menus.  Note ' ' after '?' }
PtSetString('G', ' Continua? ');
PtSetString('W', ' Velocita di scrittura? ');
PtSetString('F', ' Piu veloce? ');
PtSetString('S', ' Meno veloce? ');
PtSetString('Q', ' Basta? ');
  { Set string used in interpreting choices from ESC menus:
    Italian 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' changed to English }
PtSetString('A', 'AQGDE--HIJKLSNOF-R-TUW-XYZ');
  { Set ESC strings used to stop the dialog.  The rigid format of
    '(SI/NO)' means that 'SI' is 'stop' and 'NO' is 'don't stop'. }
PtSetString('?', ' Vuoi proprio interrompere il programma (SI/NO)? ');
PtSetString('C', ' Un momento, per piacere. ');
```

EXAMPLE: Restoring Ports System Strings to (English) Default Values:

```
    { Set string used to remind user to press Return at end of input }
  PtSetString('R', ' When you''re done, press Return. ');
    { Set strings used in waiting for user }
  PtSetString('1', ' To continue, please press the space bar. ');
  PtSetString('2', ' Please press a key so we know you are there. ');
  PtSetString('3', ' If you don''t press a key, the program will stop. ');
  { Set strings used in ESC menus.  Note ' ' after '?' }
  PtSetString('G', ' Go on? ');
  PtSetString('W', ' Writing speed? ');
  PtSetString('F', ' Faster writing? ');
  PtSetString('S', ' Slower writing? ');
  PtSetString('Q', ' Quit? ');
    { Set string used in interpreting choices from ESC menus. }
  PtSetString('A', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
    { Set ESC strings used to stop the dialog.  The rigid format of
      '(SI/NO)' means that 'SI' is 'stop' and 'NO' is 'don't stop'. }
  PtSetString('?', ' Shall we STOP (YES/NO)? ');
  PtSetString('C', ' One moment, please. ');
```

EXAMPLE: Use of PtGetString

```
  PROCEDURE SpecialAwaitUser( Special: STRING );
  { This procedure is the same as TxpAwaitUser except that it it
    displays Special when asking the user to press the space bar
    to continue.}
  VAR WaitMessage: STRING;
  BEGIN
    PtGetString('1', WaitMessage);  { Save the current message }
    PtSetString('1', Special);  { Set the new value. }
    TxpAwaitUser;  { Wait for user to press space bar }
    PtSetString('1', WaitMessage);  { Restore the original message }
  END;  {SpecialAwaitUser}
```

This procedure could be used as follows:

```
    SpecialAwaitUser( 'Tap the space bar.');
    SpecialAwaitUser( 'Now a second tap.');
    SpecialAwaitUser( 'One more time (number three) please.');
```

3.11 Miscellaneous

```
PROCEDURE ChainTo(CodeFileName: STRING);
```

ChainTo prints, in the system message port's PtLoud text mode, a "please wait a moment" message in the system message port and causes the current program to exit and starts execution of the code file specified by CodeFileName. If CodeFileName is the null string (''), ChainTo attempts to execute a standard "startup" program.

CodeFileName is a string consisting of the name one would type at the p-System main prompt line after pressing 'X' for eXecute a file. Thus, the suffix '.CODE' should not be included in CodeFileName.

```
PROCEDURE RealToString(Source: REAL; VAR Result: STRING;
                       Size, Frac: INTEGER);
```

RealToString converts the real number Source into the string Result with Frac digits to the right of the decimal point and a length of no more than Size. If Frac is 0 then the decimal point is omitted. If Frac is less than zero then the number will be returned in scientific notation with -Frac digits to the right of the decimal point. If Size is too small to represent the real then Result is set to the null string.

EXAMPLES:

RealToString(-17.25,Result,8,3) returns with Result = '-17.250'
RealToString(-17.25,Result,8,2) returns with Result = '-17.25'
RealToString(-17.25,Result,8,1) returns with Result = '-17.3'
RealToString(-17.25,Result,8,0) returns with Result = '-17'
RealToString(-1.725,Result,8,-4) returns with Result = '-1.7250'
RealToString(-17.25,Result,8,-1) returns with Result = '-1.7E1'
RealToString(-17.25,Result,8,-2) returns with Result = '-1.73E1'
RealToString(-17.25,Result,8,-3) returns with Result = '-1.725E1'
RealToString(-17.25,Result,8,-4) returns with Result = '' (the null string) because the anticipated result, '-1.7250E1', is longer (by 1 character) than the allowed maximum size of 8.

NOTES:

No check is made against the hardware's maximum accuracy.
The least significant digit returned is rounded.
Trailing zeros (to the extent specified by Frac) are not suppressed.
Zero exponents are suppressed.
Plus signs and insignificant leading zeros are suppressed.
There is no padding of the string by blanks.
All decimal points have at least one digit to either side.

```
FUNCTION FindReal(Source: STRING; VAR Result: REAL;
                  VAR Start, Size: INTEGER): BOOLEAN;
```

FindReal searches Source, starting at position Start, for the first legal real number that string may contain.

If the function finds a number, it returns TRUE after setting Result to the number's value, Start to the position of the start of the number, and Size to the number of characters found in the number.

If the function cannot find a number it returns the value FALSE after leaving Start unchanged and and setting Result and Size to indicate the reason no legal number was found:

Size	Result	Meaning
0	undefined	No digit found in the string (scan error)
>0	<>0	Numeric overflow (sign of result tells if + or -)
>0	0	Exponent too small (numeric underflow)

NOTES:
If the string contains many leading zeroes, the function may indicate that the number is too large (overflow) when this is not the case.
The function may "blow-up" if the size of the exponent in scientific notation is greater than the maximum allowable integer on the system (e.g., 1.0E123456) or if the number of digits exceedes MaxExp, the largest power of ten that the floating point hardware can represent (e.g., 1234567890123456789.012345678901234567890).

```
FUNCTION GrPixel(XWindow, YWindow: REAL;
                 VAR XPixel, YPixel: INTEGER): BOOLEAN;
```

GrPixel maps, using the CAG's window, the "real world" coordinate (XWindow, YWindow) into pixel coordinates. GrPixel returns TRUE if the given coordinate is on the screen, FALSE if not.

NOTE: the graphic screen's coordinate system has its origin (0, 0) in the lower left corner, with values of X increasing to the right and values of Y increasing upwards.

```
FUNCTION PtDetour(ControlBool: BOOLEAN; Prompt: STRING): BOOLEAN;
```

PtDetour returns ControlBool or NOT ControlBool, depending in part on the value of TxpDebug at the time PtDetour is called. If TxpDebug is FALSE, PtDetour simply returns ControlBool. If TxpDebug is TRUE, PtDetour writes in the system message port the string Prompt followed by the string " = T " if ControlBool is TRUE, " = F " if ControlBool is FALSE. A one key response is then solicited from the user. The valid responses are: backspace or rubout, which returns NOT ControlBool, or space, which returns ControlBool. Any other key generates a tone and causes the read to repeat.

PtDetour's intended use is to allow a programmer or designer to alter the normal flow of control interactively. An often used software structure takes the form REPEAT <read> <parse> <respond> UNTIL Continue, where Continue is some BOOLEAN expression that indicates when to exit the loop. Passing this expression to PtDetour would allow one, provided he/she set TxpDebug TRUE, to remain in the loop as long as desired.

```
FUNCTION PtColorCopyMode(Back, Fore: PtColorRange;
                         BWCopyMode: INTEGER): INTEGER
```

This procedure is used in conjunction with DrawBlock. Back and Fore are the line colors that are used for the background and foreground, respectively, of the pattern to be drawblocked. BWCopyMode is the mode the pattern is to be drawblocked in. PtColorCopyMode takes these three numbers and returns an integer that encodes the background, foreground, and copymode. This number is then passed to DrawBlock as the "ColorCopyMode".

4. Input / Output Options

OPTIONS	DEFAULT	ACTIVE ON
LeftAdjust, RightAdjust, Centered	LeftAdjust	Write
AskScroll, AutoScroll, DemandScroll, NoScroll	AskScroll	Read & Write
AnyCase, UpperCase, LowerCase	AnyCase	Read
SingleSpacing, DoubleSpacing	SingleSpacing	Write
ClearLine, NoClearLine	NoClearLine	Read & Write
StartLine, NoStartLine	NoStartLine	Read & Write
EndLine, NoEndLine	NoEndLine	Read & Write
Tone, NoTone	Tone	Read
Echo, NoEcho	Echo	Read
Slow, NoSlow	Slow	Write

In the above table, the option with the greatest "precedence" is given last. For example, if RightAdjust and Centered are specified in the same option set, the Centered option will take effect, and NOT the RightAdjust option. Thus, the default options are the most easily replaced. In addition, those options specified in the option set of TxpRead [3.3] or TxpWrite replace ANY option of the same group (or row in the table) for the duration of that TxpRead or TxpWrite. After the TxpRead or TxpWrite, the options specified by the most recent PtDefine [3.1] or TxpNewOptions [3.1] will be restored.

The LeftAdjust, RightAdjust, Centered options specified how each line written through the textport should be aligned within the port. The RightAdjust and Centered options also imply the following options for Write but not for Read:

Centered implies ClearLine, StartLine
RightAdjust implies ClearLine, StartLine, EndLine

However, if NoClearLine (or NoStartLine) is EXPLICITLY included in the options given as part of a call to TxpWrite, then even though Centered or RightAdjust is in effect, the line will NOT be cleared (anew line will NOT be started) at the start of the write.

The Scroll options specify the conditions under which scrolling occurs:

AskScroll: When I/O would use last line of port, tell user (in Ports unit message port, via TxpAwaitUser) to press the space bar and wait until that is done before scrolling.

AutoScroll: Scroll when I/O would go beyond end of port (without consulting user).

DemandScroll: Scroll only in response to direct command to do so from program (via a TxpScroll).

NoScroll: No scrolling allowed.
Note: If the option DoubleSpacing is in effect, ScrollSize should be set to a minimum of two.

The "Case" options (AnyCase, UpperCase, LowerCase) govern case selection during TxpRead. If the UpperCase option is specified, all alphabetic characters are converted to uppercase before being stored and (if Echo is specified) being echoed. Specifying LowerCase instructs the TxpRead routine to convert all alphabetic characters to lowercase. The AnyCase option indicates that no case conversion is to be performed.

The "Spacing" options (SingleSpacing, DoubleSpacing) govern the spacing at each new line during a TxpWrite. If the SingleSpacing option is specified, the output is given on each line. If the DoubleSpacing option is specified, the output is given with a blank line between each line written. The blank line is placed AFTER each line of output. A TxpRead is always single spaced.

ClearLine specifies that immediately before the first character is read or written, the line on which that character will appear is cleared from the current cursor position (i.e., where the character will appear) through the last position in the textport on this line. If the operation being done is a read with no echo, the current cursor position is the place where the character would appear if Echo had been specified. The cursor position is not changed by ClearLine. NoClearLine indicates that the ClearLine action is NOT to be taken.

StartLine forces the cursor to be positioned in the first column of the textport before the Start of any character I/O. If the cursor is already in this column, no action is taken. Otherwise, the cursor moved to the first column of the next row; scrolling may be performed if required. The StartLine action comes BEFORE any ClearLine action. NoStartLine indicates that the cursor is not to be moved prior to actual character I/O.

EndLine forces the cursor to be positioned in the first column of the textport after the End of any character I/O. If the cursor is already in this column, no action is taken; otherwise, the cursor is moved to the first column of the next row. NoEndLine indicates that the cursor is not to be repositioned after character I/O is complete.

Tone indicates that an audible prompt be given before TxpRead accepts entry. NoTone omits this prompt.

Echo indicates that all valid characters entered from the keyboard via TxpRead will, after case conversion as specified by the "Case" options, be displayed (echoed) on the screen. Echoed characters will be displayed in the PtEchoed text mode. NoEcho omits the displaying of the characters.

Slow indicates that the rate at which text is displayed in the port by TxpWrite may be slowed down to facilitate student comprehension. NoSlow means that TxpWrite will display text as quickly as the software and hardware allow.

5. Table of Input / Output Status Codes

Code	Set by Routines	Meaning
>0	TxpWrite	Amount of output TxpWrite was unable to print
0	TxpWrite, TxpRead, TxpNextLine, TxpGoTo, TxpPause, TxpAwaitUser, TxpESC, PtDefine, TxpDelay	No error
-1	TxpWrite, TxpRead, TxpNextLine	Unable to move to the next line after end of Input or Output when the EndLine option was in effect.
-3	TxpGoTo	During TxpGoTo, Unable to move to specified column position.
-4	TxpGoTo	During TxpGoTo, column position OK, but unable to move to specified line position.
-5	TxpRead, TxpGetKey, GrCursor	Read timed out.
-6	TxpWrite, TxpRead, TxpScroll, TxpPause, TxpAwaitUser, GrCursor, TxpGetKey, Pause, Delay	User selected coder-defined option while in Extra Services Control request.

6. Routine Name and Number Concordance

This table lists, for every Ports unit routine available to the programmer, its number as assigned by the compiler and its parameter list. The numbers may be used by the programmer to trace the sequence of execution when run-time errors occur.

```
 2  FUNCTION  TxpCursor(Active: BOOLEAN): BOOLEAN;
 3  PROCEDURE TxpTone(Pitch, Duration: INTEGER);
 4  PROCEDURE ChainTo(CodeFileName: STRING);
 5  PROCEDURE TimerSet(VAR Timer: TimerType);
 6  FUNCTION  TimerRead(Since: TimerType): INTEGER;
 7  PROCEDURE TxpESCSet(PromptMessage: STRING; Letters: Alphabet);
 8  FUNCTION  TxpESCRequest: CHAR;
 9  PROCEDURE TxpSetTime(ReadTime, StrokeTime: INTEGER);
10  PROCEDURE TxpGetTime(VAR ReadTime, StrokeTime: INTEGER);
11  PROCEDURE TxpESC;
12  PROCEDURE TxpNewOptions(NewOptions: TxpOptSet);
13  PROCEDURE PtDefine(VAR Name: Port; LeftColumn, TopOfPort,
                       WidthOfPort, HeightOfPort: REAL;
                       ScrollDepth: INTEGER; Options: TxpOptSet);
14  PROCEDURE PtDispose(VAR Name: Port);
15  PROCEDURE TxpSelect(Name: Port);
16  PROCEDURE PtSelect(Name: Port);
17  PROCEDURE TxpClrln;
18  PROCEDURE TxpClrpt;
19  PROCEDURE PtEraseAll;
20  PROCEDURE PtErase(Name: Port);
21  PROCEDURE TxpWhereAmI(VAR X, Y: INTEGER);
22  FUNCTION  TxpStatus:INTEGER;
23  FUNCTION  TxpLineLast:BOOLEAN;
24  PROCEDURE TxpNextLine;
25  PROCEDURE TxpScroll(Size: INTEGER);
26  PROCEDURE TxpGoto(X, Y: INTEGER);
27  FUNCTION  TxpWhatPort: Port;
28  FUNCTION  PtWhatPort: Port;
29  FUNCTION  TxpTicSize(NewValue: INTEGER): INTEGER;
30  PROCEDURE PtInit;
31  PROCEDURE TxpPause(HowLong: INTEGER);
32  PROCEDURE TxpDelay(HowLong: INTEGER);
33  PROCEDURE TxpGetKey(VAR Key: CHAR);
34  PROCEDURE TxpAwaitUser;
35  PROCEDURE TxpWrite(OutStr: TxpLongString; Options: TxpOptSet);
36  PROCEDURE TxpRead(VAR InStr: TxpLongString; MaxLength: INTEGER;
                      Options: TxpOptSet);
37  PROCEDURE RealToString(Source: REAL; VAR Result: STRING;
                           Size, Frac: INTEGER);
38  FUNCTION  FindReal(Source: TxpLongString; VAR Result: REAL;
                       VAR ScanStart, Size: INTEGER): BOOLEAN;
39  PROCEDURE GrWindow(XLeft, YBottom, XRight, YTop: REAL);
40  PROCEDURE GrSelect(GrPort: Port);
41  PROCEDURE GrFrame;
42  PROCEDURE GrLineTo(NewX, NewY: REAL);
43  PROCEDURE GrLine(DeltaX, DeltaY: REAL);
```

```
44  PROCEDURE GrMoveTo(NewX, NewY: REAL);
45  PROCEDURE GrMove(DeltaX, DeltaY: REAL);
46  PROCEDURE GrArc(XCrntToCenter, YCrntToCenter, InteriorAngle: REAL);
47  PROCEDURE GrSetLnMode(Ink: GrLnModeType);
48  PROCEDURE GrSetLnStyle(NewStyle: GrLnStyleType);
49  PROCEDURE GrWhereAmI(VAR NewX, NewY: REAL);
50  PROCEDURE GrStepSize(XStepUnit, YStepUnit: REAL);
51  PROCEDURE GrCursor(VAR XWanted, YWanted: REAL);
52  FUNCTION  GrPixel(XWindow, YWindow: REAL;
                      VAR XPixel, YPixel: INTEGER): BOOLEAN;
53  FUNCTION  GrClipping(Active: BOOLEAN): BOOLEAN;
54  FUNCTION  GrWhatPort: Port;
55  FUNCTION  PtDetour(ControlBool: BOOLEAN; Prompt: STRING): BOOLEAN;
56  PROCEDURE PtSetString(Tag: CHAR; AString: STRING);
57  PROCEDURE PtGetString(Tag: CHAR; VAR AString: STRING);
58  PROCEDURE PtSetCrScheme(NewSchName: PtCSName);
59  PROCEDURE TxpSetWrColors(NewMode: PtTextMode);
60  PROCEDURE GrSetLnColor(NewLnColor: PtColorRange);
61  PROCEDURE PtDefCS(Name: PtCSName; CS: STRING);
62  PROCEDURE PtDisappear(PortName: Port);
63  PROCEDURE GrBox(WhatPort: Port;
                    StartCol, StartRow, NumCols, NumRows: REAL;
                    FlashCount: INTEGER);
64  FUNCTION  PtColorCopyMode(Back, Fore: PtColorRange;
                              BWCopyMode: INTEGER): INTEGER;
```

7. Trouble Shooting

One of the unwritten rules of programming is that a program of any reasonable size is never going to be bug-free. The challenge is to reduce the number of errors in a program as much as possible. The Ports unit provides a mechanism to aid in excising the errors from a program. This mechanism is the public global boolean variable TxpDebug.

The value of TxpDebug may be changed in two ways: explicitly by the program (TxpDebug := TRUE, for example) or by the user through the calling of procedure TxpESC. The two unlisted options of TxpESC are open- and close-parenthesis. Typing an open-parenthesis when the Extra Services Control prompt is displayed causes TxpDebug to be set TRUE. Typing a close-parenthesis causes it to be set FALSE. TxpDebug is initially set FALSE by PtInit.

TxpDebug's effect on the Ports unit is to disable the "time out" condition of input routines. When TxpDebug is TRUE, all input routines assume an "infinite" read time, and will not exit until a response is entered. Exception: the asynchronous use of TxpGetKey. When TxpReadTime is 0, TxpGetKey is not affected by the value of TxpDebug.

The program may test the value of TxpDebug and (if TRUE) display information that would be useful to know during runtime. Such information might include the name of the procedure that is currently being executed, a more general section marker, the dimensions of some ports in use, the number of times a section of code has been executed, or the results of a pattern-matching algorithm on the user's last response.

What follows is a list of common problems that arise when coding a piece of software that uses the Ports unit or related underlying software. Each problem is listed with its symptoms and probable solutions.

SYMPTOM: floating point overflow after responding to a program query.

PROBLEM: the program is calling the U.C.S.D. intrinsic TRUNC with a real number that is greater than MAXINT. The real number's value was probably set by the FindReal routine (a routine commonly used by E.T.C. dialogs to locate real numbers in strings and return their value).

SOLUTION: test the value of the real number returned by FindReal against MAXINT BEFORE calling TRUNC. If the real is greater than MAXINT, set the real equal to MAXINT and then call TRUNC.

SYMPTOM: NIL pointer reference error message.

PROBLEM: a port was referenced without first being defined by PtDefine.

SOLUTION: locate the place in the code where the error message was displayed by the operating system. Note all ports being referenced in that area, and attempt to find where they are PtDefine'd.

SYMPTOM: the screen was not erased at the start of the program and either a floating point error occurred or the text written by TxpWrite appeared at an extremely slow speed.

PROBLEM: PtInit was not called before other Ports unit routines were called in the program.

SOLUTION: place in the program a call to PtInit before other Ports unit routines are called.

7.1 Error Messages

The error messages generated by the Ports unit are all of one format. All error messages are given in the Ports unit message port, which is located at the bottom of the screen. The error messages look like:

X: V=<value> R=<lower>,<upper>

where X is a character. 'V' denotes the Value that was out-of-bounds. 'R' denotes the Range of acceptable values. As a rule, X will be lowercase when the error was a purely graphical one, uppercase otherwise. These are the possible values of X:

(This group of messages is generated by PtDefine.)

```
L - the Left edge of the port is not on the screen (0..79 on most).
T - the Top edge of the port is not on the screen (0..22 on most).
W - the Width of the port is either less than one or so large
    that all of the port will not fit on the screen.
H - the Height of the port is less than one or so large that the
    entire port will not fit on the screen.
S - the Scroll height is not in the range 1..Height.
```

(This group of messages is generated by GrBox. The values given are pixel coordinates which correspond to the text coordinates.)

```
l - the left edge of the box is either not on the screen or not
    within the dimensions of the port specified.
t - the top edge of the box is either not on the screen or not
    within the dimensions of the port specified.
w - the width of the box is so large that all of the box will
    either not fit on the screen or not fit in the port specified.
h - the height of the box is so large that all of the box will
    either not fit on the screen or not fit in the port specified.
```

(The routines that generate the following messages do not use the value or range fields of the error message. The integers displayed in these fields are to be ignored.)

a - GrArc: the CAG's window is not defined.
C - The string passed to PtDefCS (after extraneous characters are removed) is not exactly 12 characters long.
c - GrCursor: the CAG's window is not defined.
D - PtDispose: the port was never defined.
i - GrWindow: the parameters given describe a window of zero area.
m - MapToWndw: the CAG's window is not defined.
N - TxpSelect, PtSelect: the named port is NIL.
n - GrSelect, PtSelect: the named port is NIL.
p - MapToPort: the CAG's window is not defined.
s - GrStepSize: the CAG's window is not defined.

8. Glossary

Adjustment - the way in which lines of text are aligned when output in a textport. There are three types of adjustment within a textport:
LeftAdjust: each text line is positioned at the extreme left edge
Centered: each text line is centered within the left and right edges
RightAdjust: each text line is positioned at the extreme right edge
These output options are discussed further in section 5.

Allocate - To request a piece of memory from the system for a specific application. The system will regard that piece as belonging to the user until the user indicates that it is free for other use (see Disposal). Allocations made by user programs are placed on the heap (see below).

Aspect Ratio - In graphics, the ratio of a picture's width to its height. In Ports, a graphport has a 1:1 aspect ratio if a line of a particular length has the same apparent length when displayed horizontally as when displayed vertically. Pictures drawn in such a graphport will be undistorted. However, the further the aspect ratio changes from 1:1, the more distorted (either flattened or elongated) th displayed picture will be. The procedure GrWindow [3.1] permits the programmer to request a coordinate system deliberately arranged to have a 1:1 aspect ratio.

Asynchronous - "without synchronization." Describes an action that doesn't wait for (is not synchronized with) any other action. EXAMPLE: an asynchronous read from the keyboard returns the most recent character pressed since the previous read operation (if any), WITHOUT WAITING for the user to press a key (so it may return no character at all).

CAG - See Currently Active Graphport

CAP - See Currently Active Port

CAT - See Currently Active Textport

CGP - See Current Graphic Point

CTP - See Current Text Position

Chaining - causing one program to execute another. The first program must terminate before the second can actually start execution.

Clipping - During graphics display, the process of "discarding" (that is, not showing) any portion of the "real world" which is not within the window (and which therefore cannot be displayed inside the graphport).

Color Scheme - All the information a port needs for color operations:
- the port's background color;
- the port's four possible line drawing colors (these are numbered starting at 0, with 0 being the port's background color);

- the port's four possible Text Fonts (each a background and foreground color) for text output.

Current Color Scheme - The current color scheme kept internally by the Ports unit. When a port is defined, its color scheme is set to the current color scheme. It is initially set by PtInit as described below, and can be changed whenever the programmer desires.

Current Graphic Point (CGP) - The point in a graphport at which the most recent graphical drawing or moving operation ended. The CGP is one of the individual attributes maintained independently by each graphport.

Currently Active Graphport (CAG) - The (single) graphport which was most recently selected for all graphic I/O (valid until another graphport is selected, which may be done at any time).

Currently Active Port (CAP) - defined to be one of two values depending on the CAG and CAT. If they are the same port, then it is the CAP as well; otherwise, there is no CAP.

Currently Active Textport (CAT) - The (single) textport which was most recently selected for all textual I/O (valid until another textport is selected, which may be done at any time).

Current Text Position (CTP) - The position a textport of the most recent text activity. The CTP is one of the individual attributes maintained independently by each textport.

Default - An action or a value that the Ports unit assumes is to be used unless a program specifies otherwise.

Dispose - To return to the system, for other use, a piece of memory which the program has been using. Memory that may be disposed of is usually contained in the heap (see below).

Dynamically Allocated Memory - Memory allocated by the program, as part of its execution, rather than by the system.

Echoing - the process of displaying a text character in a textport. The term is relevant only in terms of textual input -- the user's response may or may not be echoed during a TxpRead [3.3] at the programmer's discretion, while TxpGetKey [3.3] never echoes the character pressed by the user.

GraphPort - The graphical component of a port, consisting of a viewport and a window taken together: the viewport defines a portion of the screen and the window specifies what part of the "real world" is displayed/viewed there. The programmer uses a graphport by drawing objects in its window (that is, the objects are specified by giving their real world coordinates, such as 4 meters for a car's length) and the user views then through the graphport's viewport (where the actual length of the car's image might be only 5 centimeters). The display may have several distinct ports, all functioning

independently, each with its own viewport, window, attributes, and current point (defined below). Graphports with overlapping viewports are possible but only rarely desirable.

Heap - A free pool of memory available to the user program for making its own allocations (usually for building data structures). Under most UCSD p-System versions, the heap is organized as a stack; this is not typical of heap implementations.

Line Drawing Color - Each port has four colors available for drawing. The first of these is always the port's background color. The current line drawing color for a port is one of these four colors.

Line Drawing Mode - In order to draw a line in a port, the pixels comprising it must be set to certain colors. The line-drawing mode defines the logical operations between the port's background color and the line-drawing color which gives the new colors for the pixels. Since arcs consist of many lines, the line-drawing mode and color also applies to them.

Line-drawing Style - the form of the line when it is drawn, whether solid (line consists of contiguous pixels), dotted (line consists of every other pixel) or dashed (line consists of a sequence of sublines). Since arcs consist of many lines, line-drawing style also applies to them.

Message Port - the port that the Ports unit defines at the bottom of the screen. It is used for displaying messages such as the "To continue, please press the space bar" output by TxpAwaitUser [3.3]. No other port may be defined so that its textport overlaps the message port's textport.

Palette - The set of four colors that may be used for graphics and text at any one time. The screen background color and all color schemes must be composed of colors from these four colors. Any colors outside of the palette that are used will be mapped to a color in the palette.

Pixel - An element of a graphic picture that cannot be broken down further. It is one picture element (hence the name) or "dot" on the graphics display.

Port - a rectangular region of the screen that has two components: a textport and a graphport (each defined in this section). It is used to input and output textual and/or graphical information.

Port Background Color - The background color of each individual port, which is superimposed on the screen background color.

Private - In a unit (see below), any identifier which is not accessible to the user program, and may therefore be used only by the unit itself, is considered private to the unit. Opposite of public (see below).

Programmer - A person writing a program which uses the Ports unit.

Public - declared by a unit to be available to programs (and other units) which use that unit. For example, all the Ports routines discussed in this document are public. Opposite of private.

Routine Number - the unique number that the UCSD p-System compiler assigns to every routine in a compilation module (unit or program -- see "Unit" below). Each routine receives its number (starting from 1) when the compiler encounters its procedure / function statement; the main program is therefore procedure #1, since the PROGRAM statement is the first one the compiler encounters. When a runtime error occurs, the p-System gives the number of the routine in which the error occurred. Section 7 contains a list of all of the public Ports unit routines and their numbers.

Rubber-banding - In interactive graphics, maintaining a visible line between the interactive graphics cursor and some fixed point. The line stretches and contracts as the user moves the cursor, causing it to resemble a rubber band.

Screen Background Color - This is the color of the entire screen on which individual ports are superimposed.

Scrolling - the act of moving a textport's contents up one or more lines (cutting off those that move up past the top edge of the textport). Scrolling is handled by TxpScroll [3.7], which may be called by other routines.

Status Code - Generally, a symbol or value that indicates the condition of something. The Ports unit function TxpStatus [3.5] returns a status code (of integer type) that describes the state of the currently active textport (defined above).

Sticky Space - In order to display a string within a textport, it is often necessary for TxpWrite to break it apart (displaying the pieces on successive lines). TxpWrite breaks the line only at spaces. Also, in RightAdjust'ing or Center'ing text, TxpWrite eliminates all spaces to the right and to the left of the group of words within the string. If the programmer specifically wishes a line of text not to be broken at a certain space or leading/trailing spaces not to be eliminated by TxpWrite, the programmer may substitute a special character (reverse accent "`") for the spaces desired. TxpWrite treats the reverse accent as any other character (such as "t" or "Q") when trying to break lines apart at spaces, but the sticky space is output as a normal blank.

System Message - a message displayed to the user in the Message Port. This message can either remind the user to press RETURN when he is finished entering an answer, tell the user to press a key to continue, or give the user different options when he presses the ESC key. System messages can be displayed in any language through the use of PtGetString and PtSetString [3.10].

Text Colors (Text Fonts) - Characters, the basic components of text, are represented by a rectangular array of dots which have two colors,

called the background and the foreground. The dots representing the character itself are in the foreground color, and the rest of the character is in the background color. In this implementation of Ports, a Text Font is a combination of a background and a foreground color and so is also called a Text Color (even though it is actually 2 colors).

TextPort - the component of a port that deals exclusively with textual input and output. Textport boundaries always lie on integral text coordinates.

TextPort Option - a scalar that indicates how the Ports unit is to perform textual input or output in a textport. Textport options indicate, for instance, the adjustment (see above) of lines to output, whether the output should be double- or singlespaced, etc. Some options control whether the user's response is echoed (see above) during a read. For each situation where a particular option must be chosen, Ports has a default option (see above), so the programmer doesn't have to specify all options for all situations.

Text Writing Modes - Each port has a set of four text fonts, each of which has a foreground and a background. These are used to represent four different modes of classifying text output, and are as follows:
Echoed - input from the learner which is echoed on the screen;
Normal - normal text output;
Loud - emphasized text output;
Quiet - de-emphasized text output.
Each port has a current text writing mode which specifies one of the four text output modes described above.

Timing Out - All input routines (described in section 4.2) have a maximum amount of time allotted to them. During this time, the user may type appropriate keys and obtain the desired response. However, if the user ceases to press keys, the read will eventually give up and control is then passed back to the calling routine. This is termed "timing out." The number of seconds before a time out occurs is regulated by the internal Ports unit variable TxpReadTime, which may be set via TxpSetTime [3.8] and queried via TxpGetTime.

Unit - 1. A device within the p-System, identified by its unit number. EXAMPLE: Unit 2 is the keyboard, units 4 and 5 are disk drives. All pre-defined routines named UNIT**** deal with this type of unit.
2. UCSD Pascal has two "compilation modules": programs and units. Units contain procedures and functions, but no main program. They are therefore not executable by themselves. Their routines are only executed by Pascal programs which use them. Programs use them by declaring the unit in a USES statement. Units provide a way of developing and pre-compiling general-purpose routines which any program may then use.

User - A person who uses (runs, executes) a program which uses the Ports unit.

Viewport - A rectangular section of the display screen where graphics images are displayed. A viewport is defined by giving its location (actually, the location of its upper-left corner) on the display screen and it's dimensions (width and height). Location coordinates and dimensions for viewports are always given in terms of character (text) widths and heights.

Window - A system of coordinates applied to a graphport by the coder, specifically for the application at hand. The actual coordinate values that the coder gives correspond to the edges of the graphport.

9. Index

Keyed File System
Reference Guide
Version 2.0
16 July 1985

This software was developed by the Educational Technology Center of the University of California at Irvine supported, in part, by various federal grants.

Address comments or questions to

Alfred Bork or
Augusto Chioccariello
Educational Technology Center
University of California
Irvine, California 92717
(714) 856-6945

or

Stephen D. Franklin
Computing Facility
University of California
Irvine, California 92717
(714) 856-5154

Designed and implemented by S. Bartlett, A. Beneschan, A. Chioccariello, S. Franklin, and N. Salvador.

The Keyed File System is implemented in Pascal using the UCSD p-System (tm) developed at the University of California San Diego's Institute for Information System, under the direction of Professor Kenneth L. Bowles. The UCSD p-System is now maintained and distributed by Softech Microsystems.

Version 2.0 differs from previous versions of the keyed file system in that it supports color textports. This color implementation is principally the work of A. Chioccariello, D. Duffin, S. Franklin, and J. Meilak.

Table of Contents

1. Introduction to Keyed Files

Why Use Keyed Files

"Algorithms + Data Structures = Programs," N. Wirth of ETH

"Programs + Keyed Files = Dialogs," Anon of ETC

Keyed Files allow the programmer to remove from a dialog most of the data needed by the program (other than that supplied interactively by the user), and to keep it instead in files separate from those that contain the code expressing the logic of the program. This separation both reduces the size of the program (source and code) and facilitates modification of the program. In particular, the use of keyed files makes it easier to modify or vary the messages presented to the user, to add or change words that are recognized by the program in users' responses, and to translate the entire dialog from one natural language to another without recompiling the program's logic.

Four Critical Definitions

A "Keyed File" is a data file organized as a collection of "Messages" each of which can be individually accessed and retrieved by its "key." (By convention, the UCSD file name of a Keyed File ends with the suffix ".KFIL".)

A "Key" is a string, of 1 to 10 printable characters (space and ">" NOT allowed), which is used to identify a particular message in a keyed file.

Each "Message" in a keyed file is named/identified by its key and is made up of a sequence of items.

Each "Item" in a message is a string of 0 to 255 characters; these characters need not be printable and the "meaning" of an item or any character within it depends entirely on how it is processed by the program using the keyed file.

Display and Data Messages

Two types of messages, with different uses, can be stored in keyed files. A very common use of keyed files and the messages they contain is to specify the format and content of a display on the screen during the running of a dialog. Messages containing this sort of display information, and which are directly used by the DispMessage routine, are referred to as "Display Messages."

Display messages contain both general formatting information and information which will be displayed directly on the screen (e.g., specific text strings to be written to the screen). The general formatting information can specify the overall layout of the display (e.g., what the location and default options are for ports), timing (e.g., pauses and "wait for the user to press the spacebar"), clearing portions of the screen, etc.

Messages that contain data items that the calling program can process internally and/or display on the video screen are referred to as "Data Messages." These messages are not intended to be processed by DispMessage. Data items can be any data that a program processes except for that which the user will enter from the keyboard. The following are examples of the possible uses of Data messages: to store a list of data items that will be randomly accessed by the program; to define strings that can be used in conjunction with the String Analysis unit; to separate the logic of the program from any data that are dependent upon the natural language of the dialog.

A keyed file can contain both "Display" and "Data" messages freely mixed.

Creating Keyed Files: KFCreate Unit and Mulcher Program

There is a UCSD Pascal unit called KFCreate which exports definitions and routines which can be used to write programs which create keyed files as their output using textual or keyboard input. One such program for creating keyed files is called the Mulcher; however, other programs are possible.

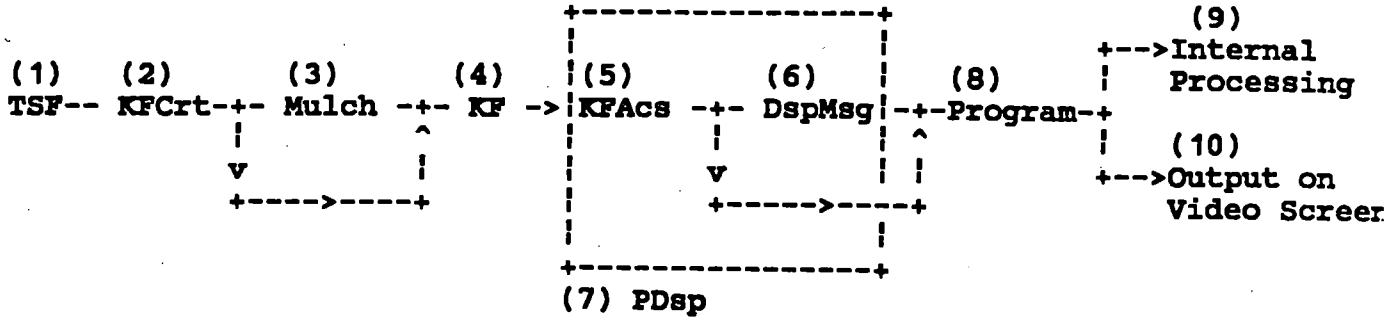
As a matter of practice, most keyed files are created using the Mulcher program operating on text files as input.

Accessing Keyed Files: KFAccess and PDisplay Units

Keyed files, the messages they contain, and the items within these messages may be accessed by using the routines contained in the UCSD Pascal unit KFAccess. These same routines are contained in the PDisplay unit which also contains the DispMessage procedure used to "display" the contents of a "display message."

2. Overall Structure

The following diagram depicts the overall structure of the "keyed file" system.



KEY

TSF	= Text Source File
KFCrt	= KFCreate Routines
Mulch	= Mulcher
KF	= Keyed Files
KFAcs	= KFAccess Routines
DspMsg	= DispMessage
PDsp	= PDisplay Routines (includes KFAccess routines and DispMessage)

1. The Text Source File (hereafter referred to as the "TSF") is a text file (call it X.TEXT) built using an editor; it may contain any number of messages, each of which may contain text and/or command lines. Individual commands and syntax are discussed later.

2. The KFCreate routines are used to create keyed files.

3. The Mulcher is a special conversion program that uses the KFCreate unit to create a Keyed File from a TSF. Mulcher does not change the TSF; it only uses it to create a different type of file.

4. The Keyed File is the file that is to be accessed at runtime. The Keyed File is identified by the keyword suffix, ".KFIL". This data file must be made available to the programs which use it prior to execution.

5. The KFAccess routines are used to access messages in the Keyed File and to retrieve one item at a time from the selected message.

6. DispMessage is a procedure which uses KFAccess routines to search for a title of a message that matches the parameter string, to retrieve, and then to display the message. The effect of "displaying" a message varies with the content of the message. Text lines in messages are directly displayed on the video screen; command lines, on the other hand, cause certain other actions to be taken by the program.

7. The PDisplay unit contains the KFAccess routines and the procedure DispMessage.

8. Program that utilizes the keyed file system.

9. The retrieved item can be internally processed by the program utilizing the keyed file system.

10. The text appearing on the screen is displayed in what shall hereafter be referred to as the CAT (Currently Active Textport). There exists only ONE CAT at any given moment during runtime. Two types of textports may act as a CAT: (1) Display Textports, which are declared within the keyed file; (2) Program-declared Textports, which are called Program Textports for short. The table given below compares and contrasts these two kinds of textports.

	Program Textports	**Display Textports**
Method for Definition	**PtDefine.**	**Program calls DispMessage with the name of a message containing a Define line.**
Method for Selection as CAT	**TxpSelect, PtSelect.**	**Program calls DispMessage with the name of a message containing a Select command.**
Method for displaying text	**TxpWrite.**	**Call DispMessage with the name of a message.**
Method for reading from a port	**TxpRead or TxpGetKey.**	**TxpRead or TxpGetKey.**
Method for saving a port	**TxpWhatPort or PtWhatPort.**	**TxpWhatPort or PtWhatPort.**
Syntax of port names	**Pascal rules for identifiers apply**	**Pascal rules for identifiers apply**

For an explanation of the Define and Select commands mentioned above, see the section on available commands.

Although DispMessage uses the Ports system, it is currently aware only of the textport aspect of its operation.

3. Creating a Keyed File

3.1 Format of the Text Source File (TSF)

A typical TSF is broken up into messages.

Messages are generally separated from one another by blank lines, but may be separated by any amount of text if each line begins with a special symbol.

A message consists of a start-of-message indicator, followed by the body of the message, followed by an end-of-message indicator.

Start of Message Indicator

A start-of-message indicator is an entire line of text with the format:

<<message-name>> command sequence

where "message-name" or "key" is any sequence of characters except space and ">", and the command sequence immediately after the message name is optional.

The first ten characters of the name are stored in the Keyed File as the message name. Consequently, no two message-names may have the same first ten characters.

The case of any letter in a "message name" is, by default, semantically irrelevant to the Keyed File System. That is "GOODBOY", "GoodBoy", "goodboy" are the same "key". However, since the unit KFCreate and KFAccess allow to overwrite the default, special application programs could create keyed files which are sensitive to the case of the "key".

NOTE: The Mulcher always creates keyed files whose keys are case insensitive.

Body of the Message

The body of the message consist of one or more command lines, and text lines.

A **command line** is an entire line of text with the format:

\\ command sequence

or

\\ command sequence -- comment

The double backslash "\\" signals a start of a command line.

"Command sequence" is a sequence of 0 or more commands separated by semicolons. A semicolon is allowed, but not required, after the last command on a line. Any number of spaces greater than 0 may also appear between the commands. The available commands are described in section 3.2 .

If two hyphens ("--") appear on a command line, they and all text following them are taken as a comment and ignored by the Mulcher. If there are no actual commands on the line (i.e. the line begins with "\\ --") the entire line is considered a comment.

EXAMPLE:

```
<<Message1>> 1SPACE; CENTERED
...
```

is equivalent to

```
<<Message1>>
\\ 1SPACE; CENTERED
...
```

or to

```
<<Message1>>
\\ 1SPACE
\\ CENTERED
...
```

A **text line** is a line which does not begin with \\ or <<.

End of Message Indicator

An end-of-message indicator is an entire line of the form:

<<END>>

or

<<END Message1>>,

where the message-name (in this case Message1) must match the name given the message at its beginning. If it does not, a warning will be generated at Mulch-time.

All command or start/end of message lines (i.e., those starting with \\ or <<) MUST start at the extreme left margin (column 1). No spaces or other characters are allowed between the left margin and a line that does not contain actual text.

NOTE:

It is possible to have comments between messages using the following format:

\\-- comment

The Mulcher will treat this line as if it had been blank.

Any text line between messages that does not have this format will generate a warning at Mulch-time but is otherwise ignored.

3.2 Available Commands

DATA - Indicates a data message, and tells the Mulcher that the following message is to be placed in the Keyed File without changing the text in any way. Each line of the message becomes an item and can be retrieved by the calling program using KFGetItem. All special characters (including "\", "\\", and "--") are treated as literal data, not as normal command symbols.
IMPORTANT: This command must be the first command of the message.

EXAMPLE: For purposes of this example, each item in the message is on its own line. Here, each item contains a list of synonymous expressions of the same level of praise and can be extracted from the message and processed by the calling program.

```
...
<<Praise>> DATA
\ Outstanding \ Excellent \ Superb \ First Class \
\ All right \ Fine \ Good \
\ Not bad \ That will do \ OK \
\ Not so good \ Not good enough \
\ Ouch! \ Poor \ Terrible \
<<END>>
...
```

LAYOUT - Tells the Mulcher that the lines between it and the next command line or the end of the message are display textport definition lines.
IMPORTANT: This command MUST be the only command on its line (it can be on the same line as the message name), and must be the first command of the message.
NOTE: The LAYOUT and the DATA commands are mutually exclusive - they cannot both be used in the same message.

EXAMPLE:

```
<<MessName>>
\\LAYOUT  --Note that this immediately follows the start of the message!
  SetCrScheme(PtCs1);
  Define(Port1, 57.5, 13.5, 14, 5, 3, [StartLine, ClearLine,
        EndLine, NoTone, NoSlow]);
  SetCrScheme(PtCs2);
  Define(Port2, 5, 12, 35, 6, 2, [RightAdjust, NoTone]);
  Define(Port3, 47.5, 4.5, 28, 3, 1, [Centered, NoSlow]);
\\SELECT(Port1)
<text> or <commands>
<<END>>
```

The syntax for Define lines is the same as that used for PtDefine, and the commands may take up more than one line if needed. The indentation is not

needed, although it does make the block easier to read.

The syntax for SetCrScheme command is the same as that used for PtSetCrScheme. That is, the allowable parameters for SetCrScheme are PtCSSystem, PtCSHelp, PtCSSummary, PtCS1, PtCS2, PtCS3, PtCS4, PtCS5. For a description of color schemes see the Ports Unit Reference Guide, section 3.2.
Note that in the above example Port1 will have color scheme PtCS1 associated with it, while Port2 and Port3 will have color scheme PtCS2 associated with them. If no SetCrScheme appears in a layout block, then all the ports defined in that block will have the most recent color scheme (set by a SetCrScheme command in a previous layout) associated with them. If there is no previous SetCrScheme command in the TSF, then all the ports defined will be given the default color scheme. Although a color scheme from a previous layout block can "carry over" to the next layout block, it is good programming style to call SetCrScheme to set the color scheme at the beginning of each Layout command.

NOTE:
A semicolon is required at the end of a Define line, EVEN if it is the last line of the block. Display textport names follow the same Pascal rules for identifiers that program textport names do.

BREAK	**- Interrupts the current display message and returns to the calling procedure, which may later return to the same message. Usually used to display graphics in the middle of a message, or to display a variable string. IMPORTANT: this command MUST be the last command on its line.**

Commands to set Ports options:

LEFT	**- Sets the LeftAdjust option: DispMessage will use the LeftAdjust Ports option when displaying succeeding lines, until a contradictory command (in this case, RIGHT or CENTERED) appears.**
RIGHT	**- Sets the RightAdjust option.**
CENTERED	**- Sets the Centered option.**
SLOW	**- Sets the Slow option.**
FAST	**- Sets the NoSlow option.**
1SPACE	**- Sets the SingleSpacing option.**
2SPACE	**- Sets the DoubleSpacing option.**

NOTE: The default options are LEFT, SLOW, and 1SPACE.

Commands to call Ports routines:

```
SELECT(xxx)       - Calls TxpSelect, which activates the display
                    textport named xxx.
                    IMPORTANT: this command MUST be the first command
                    on its line.

DISAPPEAR         - Calls PtDisappear(TxpWhatPort).
DISAPPEAR(ALL)    - Calls PtDisappear on all the ports defined in the LA
DISAPPEAR(xxx)    - Calls PtDisappear(xxx).

ERASE             - Calls PtErase(TxpWhatPort).
ERASE(ALL)        - Calls PtEraseAll.
ERASE(xxx)        - Calls PtErase(xxx).

DELAY(n)          - Calls TxpDelay(n).  n is an unsigned integer.
PAUSE(n)          - Calls TxpPause(n).  n is an unsigned integer.

WAIT              - Calls TxpAwaitUser.

ECHOED            - Calls TxpSetWrColors(PtEchoed).
LOUD              - Calls TxpSetWrColors(PtLoud).
NORMAL            - Calls TxpSetWrColors(PtNormal).
QUIET             - Calls TxpSetWrColors(PtQuiet).
                    NOTE: The default writing mode is NORMAL.
```

Parameters must be separated from their commands by blanks. In the interest of readability however, it is suggested that the user might enclose parameters in parentheses. Parentheses ARE REQUIRED around the Define statement parameters.

3.3 Controlling Line Breaks

The coder can control where a text line will be broken by using the phrase marker "\" (backslash) and the sticky space "`" (reverse accent) character, described below. When a text line is displayed, each character in it is written on the screen EXCEPT the phrase marker and the sticky space.

The backslash has the following meanings:

> If it is not the last character in a text line, the backslash will act as a "weak point" in the line. This means that, if the line proves to be too long to fit in a given textport, DispMessage breaks the line at that point, treating the phrases on either side as separate groups of words, each to be kept intact if possible. If the port is wide enough, the backslash will be ignored when displaying the text.
>
> If it is the last character in a text line, "\" will not only act as a "weak point", but will also be taken as a command NOT to move to a new line in the port upon reaching the end of the text line.

The reverse accent acts as a kind of "sticky space" which is useful for binding printed formulas such as X + Y + 57 = _____ together in a port where the width might not be adequate to print both the formula and a phrase on the same line. The "sticky space" character is not processed by DispMessage nor the Mulcher; only by TxpWrite. For more information on the "sticky space" character, refer to the Ports Unit Reference Guide, section 4.3.

3.4 Organization of the Text Source File

Although the placement of messages within the TSF is mostly a matter of convenience, there are certain restrictions that must be followed. Also, the display will operate more smoothly (with fewer disc accesses) if messages which are to be displayed consecutively are consecutive in the TSF (though this is not always possible).

Messages are generally separated from one another by blank lines (as many as needed for clarity), but may be separated by any number of comment lines.

A typical TSF is broken up into groups of messages, each associated with a "leading" message containing a Layout command ("leading" here refers to the fact that the message containing the Layout command is positioned before the rest of the messages in the group).

All messages that contain "select" command lines and follow a message containing a block of Define lines are said to "belong" to that message. The "select" commands that appear in messages MUST reference ONLY those ports defined in the leading message they belong to.

The reason for this is that after the Mulcher processes a group of messages and reaches another Layout command, those display textport names and parameters used in the previous group immediately become inaccessible to the Mulcher, and the new display textport parameters are written over the old ones. It is important that the user structure a TSF so that this error does not occur.

All Defines following the most recent SetCrScheme command will copy that color scheme into all following port definitions until the next call to SetCrScheme. The current color scheme carries across messages and layouts.

It is IMPERATIVE that the user avoid the following TSF organizational error.
PLEASE READ THIS EXAMPLE CAREFULLY!
Suppose the following situation exists in a TSF:

A set of messages, Set A, is followed sequentially by another Set B...

```
<<Amess1>> LAYOUT
  SetCrScheme(PtCs1);
  Define(Aport1, . . .);
  Define(Aport2, . . .);
\\SELECT(Aport1)
<text>
<<END Amess1>>

<<Amess2>>
\\SELECT(Aport2)
<text>
<<END Amess2>>

<<Amess3>>
<text>
* \\SELECT(Bport1)              (* ILLEGAL *)
<text>
<<END Amess3>>

<<Bmess1>> LAYOUT
  SetCrScheme(PtCs2);
  Define(Bport1, . . .);
  Define(Bport2, . . .);
\\SELECT(Bport1)
<text>
<<END Bmess1>>

<<Bmess2>>
\\SELECT(Bport2)
<text>
<<END Bmess2>>

<<Bmess3>>
<text>
* \\SELECT(Aport1)              (* ILLEGAL *)
<text>
<<END Bmess3>>
```

Suppose further that the Set A layout is activated at runtime before the Set B layout. In Set A, "Amess2" and "Amess3" belong to "Amess1" because they follow "Amess1" SEQUENTIALLY in the TSF and contain "select" lines. The same holds for the messages in Set B -- they belong to "Bmess1." In this case, the selects in Amess1, 2, and 3 may only reference the ports listed in the layout of "Amess1." The same is true for the Set B. Therefore , both of the starred selects are ILLEGAL, and will cause an error during mulching.

3.5 The Mulcher Program

The Mulcher is a commonly used program that uses a unit, KFCreate (described below), to create a Keyed File from a text source file. Unlike the KFCreate unit, the Mulcher recognizes commands to the PDisplay unit. As a consequence, the KFCreate unit can be used to create only display messages which DO NOT utilize the various commands described in section 3.3 (e.g., messages that contain text lines only). The Mulcher, on the other hand, is able to create both display messages that do or do not contain command sequences, and data messages.

Upon executing the Mulcher, the name of the text source file will be requested. The .TEXT suffix need not be typed as the Mulcher appends it automatically. The Mulcher program then requests the name of the keyed file it is to create. If a '$' is entered, then the name of the keyed file created will have the same name as the TSF. The .KFIL suffix need not be typed.

The Mulcher interrupts mulching if the user press "ESC" or to announce errors or warnings. The list of syntax errors detected by the Mulcher are listed in section 6.C.

It should be noted that the Mulcher ALWAYS creates keyed files whose keys are case insensitive.

WARNING:

The Mulcher programs creates a temporary file on the prefixed volumes. Enough disk space on the prefixed volume is therefore needed.

3.6 The Unit KFCreate

The unit KFCreate contains routines for creating a keyed file.

These are the declarations that KFCreate makes available to the program or unit using it.

```
CONST KeyLength = 10;
      MaxItemSize = 255;
TYPE Key = STRING [KeyLength];
     STRING255 = STRING [255];
     ItemStr = STRING [MaxItemSize];

VAR KFErrorCreate: RECORD
    { These error flags are set every time the program calls
      a KFCreate routine. }
      NoError,                { TRUE if no error (none of the other
                                flags is TRUE) }
      OpenKeyedFile,          { could not create keyed file }
      OpenTempFile,           { could not create temp file }
      NoRoom,                 { no room on disk }
      CloseFile: BOOLEAN;     { could not close keyed file }
      END;
    KeyCaseInsensitive: BOOLEAN;
    { Determines whether or not the key names are to be case insensitive.
      Default is TRUE.}
```

```
FUNCTION KFStartCreate (KFName: STRING255) : BOOLEAN;
```

Sets up for the creation of the keyed file named in KFName. Returns FALSE if the file cannot be created (wrong volume on line or some other IO error), TRUE otherwise.

```
PROCEDURE KFNewMessage (MessageName: STRING255);
```

Starts a new message in the keyed file being created.

```
PROCEDURE KFNewItem (Item: ItemStr);
```

Adds Item to the message currently being assembled. Does nothing if KFNewMessage has not been called since the last KFStartCreate.

```
PROCEDURE KFEndCreate;
```

Finishes the creation of the keyed file.

```
PROCEDURE KFAbortCreate;
```

Purges the current keyed file. Necessary if the creation is not to be completed, or if KFStartCreate will be called again.

To create a keyed file, the routines are called by the Mulcher as follows:

```
KFStartCreate
<zero or more message creations>
KFEndCreate
```

where a message is created with the following sequence of calls:

```
KFNewMessage
KFNewItem (zero or more times)
```

The order of calls to KFNewMessage determines the sequence of messages in the keyed file. Similarly, the order of calls to KFNewItem determines the sequence of items in a message.

KFCreate cannot create more than one keyed file at a time. If KFStartCreate is called twice to create two keyed files, there must be a KFEndCreate between them to finish the first one before the second one is started. KFAbortCreate can be used in place of KFEndCreate; KFAbortCreate will abort the creation without saving the keyed file.

4. Accessing a Keyed File

To access a Keyed File, the program must use the unit PDisplay, along with Ports, which PDisplay uses. The simplest USES statement the program can have is therefore:

```
USES  Ports, PDisplay;  (* note the spellings of BOTH units *)
```

Other units may, of course, be listed as they are needed. The only restriction is that Ports must be listed somewhere before PDisplay.

The PDisplay unit contains the DispMessage procedure and the KFAccess unit. Procedures and functions belonging to the KFAccess unit are prefixed KF.

These are the declarations that PDisplay makes available to the program (or unit) using it. READ THEM CAREFULLY: these identifiers may not be redeclared by the program.

```
CONST KeyLength = 10;
      MaxItemSize = 255;
      MaxDispPorts = 16; { maximum number of display ports allowed }

TYPE Key = STRING [KeyLength];
     ItemStr = STRING [MaxItemSize];

VAR KeyCaseInsensitive: BOOLEAN
{ Determines whether or not the access unit should be sensitive to the
  case of the key names.  This variable should be set FALSE (thereby
  making the key names case sensitive) ONLY if it was set FALSE while
  the KFCreate unit was creating the keyed file.  By default, this
  variable is TRUE.}
```

4.1 Selecting a Keyed File

Before a keyed file can be accessed, it must be opened with a call to KFOpen. Only an open file can be accessed. When the program is done with the file it must call KFClose. (NOTE: Only one keyed file at a time can be active).

```
FUNCTION KFOpen (FileName: STRING; BlocksInBuffer: INTEGER) : BOOLEAN;
```

Attempts to open the file FileName; it returns FALSE if the file cannot be accessed (file not present or not enough memory), TRUE otherwise. KFOpen cannot open more than one keyed file at a time (i.e., only one keyed file can be active at any one time). FileName must specify the COMPLETE name of the keyed file including the .KFIL suffix. BlocksInBuffer is the number of blocks of the keyed file to keep in memory at once. At runtime, a check is made to see if enough memory is available to allocate the number of blocks specified. If not, then KFOpen allocates as many blocks as memory will allow.

EXAMPLE:

```
...
IF KFOpen (Filename, BlocksToKeepInMemory) THEN BEGIN
    ...
    {body of program}
    ...
  END
ELSE DealWithMissingMessageFile;
...
```

```
PROCEDURE KFClose;
```

Closes the currently open Keyed File. It MUST be called before the program attempts to change from one keyed file to another. KFClose also disposes of the memory allocated by KFOpen.

4.2 Using Display Messages

```
PROCEDURE DispMessage (MessName: STRING);
```

DispMessage selects and displays the message called MessName. If there is no such message as MessName in the current keyed file, DispMessage prints a message using the current TextMode and the current position of the CAT, of the form "<<MessName>> not found".

Within the message, DispMessage processes the keyed file in the order in which the lines appear in the original TSF. (NOTE: DispMessage accesses only the Keyed File and never the TSF.) The manner of processing of each line depends on whether it is a text line or a command line.

Text Line: DispMessage displays the text in the CAT as it appears in the TSF. Upon reaching the end of a text line, DispMessage moves to a new line in the port, unless the text line ends with a phrase marker.

Command Line: The Display Routine processes the commands from left to right.

The commands PAUSE, DELAY, ERASE, SELECT, and WAIT are executed immediately.

The commands (ECHOED, NORMAL, LOUD, QUIET) set the writing color of the CAT, which is used when displaying text lines.

The commands (LEFT, RIGHT, CENTERED, 1SPACE, 2SPACE, SLOW, FAST) are stored and used when displaying text lines. If a command contradicts an earlier command (such as RIGHT contradicting CENTERED), the earlier command is cancelled and the later command remains in effect until it is cancelled by a still later command (whether it is in the same or a later message), or until the end of the message. WARNING: It is also possible, but not recommended, to have TxpOptions from the main program work on a message in the keyed file).

The BREAK command causes the routine to immediately return to the caller. To display the remainder of the message, call DispMessage('').

WARNING: Do NOT try to display any other message before displaying the remainder of the message containing the BREAK command.

At Mulch-time, the LAYOUT command tells the Mulcher that the following lines, until the next \\ or until the end of the message, are display textport definitions or color scheme settings. Mulcher has the capacity to define up to 16 display textports per layout, and allows an unlimited number of layouts per TSF. A table is made of the textport parameters for DispMessage to use at runtime. When DispMessage encounters a layout command in a message, that layout is said to become "active." A layout remains active until another layout command is encountered or until the end of the program is reached.

NOTE: At run time the message containing the layout command MUST be displayed by DispMessage before any messages accessing ports defined by that layout are displayed. This is because the ports are actually defined at run-time

by the DispMessage routine when it encounters a message that has a layout command in it. If a message is displayed before the corresponding layout message is displayed, the text of the message will appear in some other port that was previously defined, yielding undesireable effects.

EXAMPLE:

```
<<SetUp>> LAYOUT
  SetCrScheme(PtCS1);
  Define(APort, . . .);
\\SELECT(APort)
<text>
<<END SetUp>>

<<Intro>>
\\SELECT(APort)
<text>
<<END Intro>>
```

Now, if DispMessage('Intro') is executed at run-time before DispMessage('SetUp'), the SELECT statement in "Intro" won't have "APort" defined to select, and consequently, the text will appear in some unpredictable area on the screen.

4.3 Accessing Items Within a Message

The routines in the KFAccess unit are used to access keyed files, and to retrieve the messages they contain, and the items within these messages. The KFAccess routines work independently of the DispMessage procedure.

Once a message name is known to the program, KFSelectMessage and KFGetItem are used to retrieve items from the message. KFSelectMessage is called first to "select" any message for item retrieval. Subsequent calls to KFGetItem will return the successive items in the message, starting with the first. Items retrieved from a data message can be further processed by the calling program andor display the items on the video screen. KFEndOfMessage is used to determine whether all the items in the message have already been retrieved.

The following routines, in addition to KFOpen and KFClose described above, comprise the KFAccess unit:

FUNCTION KFFirstMessage (VAR MessageName: Key) : BOOLEAN;
Sets MessageName to the name of the first message in the file. Returns FALSE if there is no first message, i.e. the file is empty.

FUNCTION KFNextMessage (VAR MessageName: Key) : BOOLEAN;
Sets MessageName to the name of the next message in the file. The name returned is the name of the message after the message last looked at by KFFirstMessage, KFSelectMessage, or KFNextMessage. The function result is TRUE if there is a next message, FALSE otherwise, FALSE if none of the above listed routines have been called previously.

NOTE:
KFFirstMessage and KFNextMessage do NOT select a message; they only provide names of messages which can be used as arguments to KFSelectMessage.

FUNCTION KFFindMessage (MessageName: Key) : BOOLEAN;
Returns TRUE if the message named in MessageName is in the file, FALSE otherwise. KFFindMessage does not select a message and does not affect the result of KFNextMessage.

PROCEDURE KFSelectMessage (MessageName: Key);
Selects the message named in MessageName as the currently active message. Nothing happens if MessageName is not in the file.

PROCEDURE KFGetItem (VAR Item: ItemStr);
Sets Item to the next item in the currently active Message. Sets Item to the null string if there are no more items or if no message has been selected.

FUNCTION KFEndOfMessage : BOOLEAN;
Returns TRUE if there are no more items in the currently active Message (all items have been retrieved), FALSE otherwise. Undefined if no active message has been selected.

5. Glossary of Terms

Active Layout-A layout residing in a message that has been displayed. It remains active until another message containing a Layout command is displayed. NOTE: Only one active layout exists at any time.

Backslash-See phrase-marker. See also double backslash.

Data Message-A message (as defined below) which contains data that may be processed in a variety of ways by a dialog.

Define-A special command in a keyed file that defines a Display Textport. A set of Defines must be preceded by a Layout command. Each Define has the same format as PtDefine.

Display Message-A message (as defined below) intended to be accessed and displayed directly by DispMessage.

Display Textport-A Textport which is defined in a keyed file. Display Textport names and parameters are known only to the Mulcher and the DispMessage routine, consequently, the program has no control over them. The program may display text in these textports as long as the Display Textport has been selected in the keyed file or by indirect means in the program before the program attempts to write in it.

DispMessage-A procedure included in the PDisplay unit that searches the keyed file for the message name that matches the parameter string and displays that message.

Double Backslash ("\\")-A symbol that signals the start of a command line. This symbol must appear at the far left margin of a line (in column 1).

Double Hyphen ("--")-A symbol that signals the start of a comment. If this appears on a line that begins with << or \\, the Mulcher considers it and all of the following text (until the end of the line) a comment, which will not be written into the keyed file.

Item-A string of 0 to 255 characters which is a component of a message (either text lines, command lines, or data lines) in a keyed file. The characters in an item need not be printable and the meaning associated with an item or any of the characters in it will, in general, depend on the routine which is processing it.

Key-The name of a message in a keyed file. It is a string of 1 to 10 characters. Any printable characters are allowed except space (" ") and ">".

Keyed File-A data file organized as a collection of named messages each of which can be individually accessed and retrieved by its name. The suffix ".KFIL" is used to distinguish a keyed file from other types of files.

Message-A named sequence of items. Each item is itself a string of 0 to 255 characters but the meaning associated with each item depends on the routine which is processing it.

Message name-Same as "Key" as defined above.

Mulcher-A program that uses a TSF to create a keyed file. The Mulcher is also known as Text-To-KeyedFile (TTK), and Muncher.

Phrase-Marker ("\")-A symbol that can act as a command not to move to a new line, and/or as a "weak point" in a line, causing the line to separate at that point if the Display Textport is not wide enough to accomodate the entire line. The implementation of the phrase-marker depends on its position in the line.

Program Textports-Textports that are defined in the program. They may display messages from a Keyed File as long as the messages do not contain a Layout command.

Sticky Space ("‘")-A symbol that is printed as a space at runtime and acts to bind words or characters together in case the text line is longer than the width of a port.

TSF (Text Source File)-A text file intended to be processed by the Mulcher program (defined above) to produce a keyed file containing "Data Messages" and "Display Messages."

6. Appendices

6.A Format of Keyed Files

Overall Structure

The smallest structural unit in the Keyed File system is the item. An item is simply a string of any length from 0 to 255.

The next smallest unit is the message. A message is a named sequence of zero or more items. Items within any message can be retrieved only in sequence; that is, the third item of a message cannot be retrieved before the first and second items are retrieved. Each message carries a name, or "key"; the key is a string of length 1 to 10.

The largest unit is the keyed file. A keyed file is a file which contains a sequence of zero or more messages. Messages within a keyed file can be accessed sequentially; individual messages can also be accessed randomly, by key.

The keyed file is divided into 2 parts: the directory and the data portion.

Directory

The directory occupies as many blocks as necessary to store information about each message in the keyed file. The unused portion of the last block is left blank. A directory block may contain up to 32 entries of 16 bytes each. The format of these is discussed later.

The very first record in the directory is 12 bytes long and contains the following fields:

Field	Starting byte	Size in bytes
Usage a packed array of 8 characters not currently used.	0	8
FirstMessageBlock The first block of the data portion. The directory occupies 0..FirstMessageBlock-1 and the data occupies FirstMessageBlock..End of file	8	2
NumOfMessages The number of messages in the file.	10	2

Each message in the file has one corresponding directory entry, which is made up of the following fields (starting numbers relative to 0):

Field	Starting byte	Size in bytes
Key name The first 10 characters of the message name. If the message name is less than 10 characters, trailing blanks are added.	0	10
Block number on which the first item of the message starts.	10	2
Byte number within the block indicating the start of the first item (range = 1..512)	12	2
Number of items in the message	14	2

The above numbers are in integer form.

Data Portion

The data portion of the file consists of a series of items. The directory entry will point to a character within a block as the start of an item. Each item consists of one length byte {stored as CHR(n) where n is the length of the string}. The byte immediately following this item is the length byte of the next item, and so on. There are no markers to indicate the start or end of messages, as none are needed.

The data portion starts at the very first byte of block number FirstMessageBlock and extends indefinitely without regard to block boundaries. (Thus, an item may cross a block boundary.)

An item in the data portion, X, can have one of six formats:

1) Command

Character 1 of X is the header, '1'. The remaining characters of X give the commands to be executed:

```
'A': TxpAwaitUser;
'B': PtErase(TxpWhatPort);
'C': TxpPause (n).  The character following the 'C' gives the
       argument to TxpPause by the following formula:
       ORD (char) = ORD ('0') + n;
       { ORD (char) = n; }
'D': TxpDelay (n), where n is given in the same fashion as
       with 'C';
'E': TxpSelect (port n), where n is given in the same manner
       as with TxpPause.  Port n is defined by a Define item.
'F': PtErase (port n), n is stored in the above manner.
'G': PtEraseAll.  Erases the entire screen.
'H': TxpSetWrColors (n).  n is mapped as follows:
0 = PtEchoed; 1 = PtNormal; 2 = PtLoud; 3 = PtQuiet
'I': PtDisappear(TxpWhatPort);
'J': PtDisappear(port n), n is stored in the above manner.
```

2) Option & Text

Character 1 of X is the header, '2'. Characters 2 through 4 are the option set which is to be used with TxpWrite. (The internal format is stored in the string.) The rest of X is a phrase to be written.

3) Text only

Character 1 of X is '3'. The rest of X is a phrase to be displayed. The options used are the same as the last type 3 item.

4) Break

X is the string '4'. A \\ break command is executed.

5) Define

Character 1 of X is '5'; character 2 is CHR (n), where n is the number of the port to be defined (1..16). The remainder of X is the contents of the textport record, in an internal format.

6) Null

X is the null string, ''. This is ignored by the display routine. It is generated when "\\ break" is the last thing in the message and is present so that DispMessage can tell that this is not the end of the message. Otherwise it will think that the host program called DispMessage ('') after the message was finished.

6.B Syntax Diagram for the Keyed File System

```
KEYED FILE ::= { <outside line> ¦ <message> }

MESSAGE ::= "<<" <messagename> ">>"
            ( <data block> ¦ ( <newline> "\\" <data block> ) ) ¦
            (
              (
                ( <layout block> ¦ ( <newline> "\\" <layout block> ) ¦
                ( <command sequence> ( <comment> ¦ <newline> ) ) ¦
                <comment> ¦
                <newline>
              ) <message body> <newline>
            )
            "<<" "END" [ <messagename> ] ">>" <newline>

DATA BLOCK ::= "DATA" [ <comment> ¦ <newline> ] { <text line> }

MESSAGE BODY ::= { <text line> ¦
                   ( "\\" <command sequence> ( <comment ¦ <newline> ) ) }

COMMAND SEQUENCE ::= [
                       <select> [ ";" <COMMAND SEQUENCE> ] ¦
                       <command> [ ";" <COMMAND SEQUENCE> ] ¦
                     ] [ <break> ]

COMMAND ::= "LEFT" ¦ "RIGHT" ¦ "CENTERED" ¦ "SLOW" ¦ "FAST" ¦
            "1SPACE" ¦ "2SPACE" ¦ "WAIT" ¦ "ECHOED" ¦ "NORMAL" ¦
            "LOUD" ¦ "QUIET" ¦
            ( "DISAPPEAR" [ "(" ( <portname> ¦ "ALL" ) ")" ] ) ¦
            ( "ERASE" [ "(" ( <portname> ¦ "ALL" ) ")" ] ) ¦
            ( "PAUSE" "(" <unsigned integer> ")" ) ¦
            ( "DELAY" "(" <unsigned integer> ")" )

SELECT ::= "SELECT" "(" <portname> ")"

BREAK ::= "BREAK" ( <comment> ¦ <newline> )

LAYOUT BLOCK ::= "LAYOUT" [ <comment> ¦ <newline> ]
                 { <definition> ¦ <color scheme> }
```

```
DEFINITION ::= "DEFINE"          [ <comment> ¦ { <newline> } ]
               "("               [ <comment> ¦ { <newline> } ]
               <portname>        [ <comment> ¦ { <newline> } ]
               ","               [ <comment> ¦ { <newline> } ]
               <unsigned real>   [ <comment> ¦ { <newline> } ]
               ","               [ <comment> ¦ { <newline> } ]
               <unsigned real>   [ <comment> ¦ { <newline> } ]
               ","               [ <comment> ¦ { <newline> } ]
               <unsigned real>   [ <comment> ¦ { <newline> } ]
               ","               [ <comment> ¦ { <newline> } ]
               <unsigned real>   [ <comment> ¦ { <newline> } ]
               ","               [ <comment> ¦ { <newline> } ]
               <unsigned real>   [ <comment> ¦ { <newline> } ]
               ","               [ <comment> ¦ { <newline> } ]
               <TxpOptSet>       [ <comment> ¦ { <newline> } ]
               ")"               [ <comment> ¦ { <newline> } ]
               ";"               [ <comment> ¦ { <newline> } ]

COLOR SCHEME ::= "SETCRSCHEME" "(" <scheme name> ")"

OUTSIDE LINE ::= <newline> ¦ ( "\\" <comment> )

COMMENT ::= "--" <text line>

PORTNAME ::= "A.."Z" ¦ "a".."z" { "A.."Z" ¦ "a".."z" ¦ "0".."9" }

MESSAGENAME ::= { "!".."=" ¦ "?".."~" }
                ( All printable ASCII except " " & ">" )

TEXT LINE ::= { " ".."~" } <newline>
              ( All printable ASCII characters )

SCHEME NAME ::= PtCSSystem ¦ PtCSHelp ¦ PtCSSummary
              ¦ PtCS1 ¦ PtCS2 ¦ PtCS3 ¦ PtCS4 ¦ PtCS5

UNSIGNED REAL ::= <unsigned integer> "." <unsigned integer>

SIGNED INTEGER ::= [ "+" ¦ "-" ] <unsigned integer>

UNSIGNED INTEGER ::= <digit> { <digit> }

DIGIT ::= "0".."9"
```

```
TXPOPTSET ::= "[" { <newline> } ( "]" |
                                    (
                                      <TxpOption>
                                      { <newline> }
                                      { "," { <newline> }
                                        <TxpOption>
                                        { <newline> }
                                      } "]"
                                    ) )

TXPOPTION ::= "LEFTADJUST" | "RIGHTADJUST" | "CENTERED" |
              "ASKSCROLL" | "AUTOSCROLL" | "DEMANDSCROLL" | "NOSCROLL" |
              "ANYCASE" | "UPPERCASE" | "LOWERCASE" |
              "SINGLESPACING" | "DOUBLESPACING" |
              "NOCLEARLINE" | "CLEARLINE"
              "NOSTARTLINE" | "STARTLINE" |
              "NOENDLINE" | "ENDLINE" |
              "TONE" | "NOTONE" |
              "ECHO" | "NOECHO" |
              "SLOW" | "NOSLOW"
```

6.C Error and Warning Messages from the Mulcher

This list gives the error and warning messages output by the program. Warnings are given when something is wrong, or something might be wrong, but the program is able to make an assumption and still produce a meaningful keyed file. Errors are given for more serious problems; although a keyed file will be created (in most cases), it will probably not be meaningful.

ERROR MESSAGES

**** Program error - missing END ****
The Mulcher couldn't find an expected "END". A program bug.

**** Program error - missing SOM-marker ****
The Mulcher couldn't find "<<" that it knew was supposed to be there. A program bug.

"DATA" must be first thing in message
If a message contains a DATA command, there must not be any text or commands before the DATA command.

"Define" expected
Syntax error in port definition.

"(" expected
Syntax error in port definition or command line.

"," expected
Syntax error in port definition.

"[" expected
Syntax error in port definition.

")" expected
Syntax error in port definition or command line.

";" expected
Syntax error in port definition or command line.

Cannot close file
The keyed file could not be closed, due to some IO error.

Extra "("
Syntax error in port definition or command line.

Extra commands on line – ignored
The LAYOUT or DATA command must be the last thing on the command line. Anything after the command is not checked for validity.

Height is zero
In a port definition, the height is not allowed to be zero.

Integer expected
The word after PAUSE or DELAY was not a valid unsigned integer, or there was nothing after PAUSE or DELAY; or there was a syntax error in a port definition.

Identifier expected
Syntax error in port definition.

Layout must be first thing in message
If a message contains a layout, there must not be any commands or text before the layout.

Missing ">>"
The program expected ">>" after a message name or END.

Missing ")"
Syntax error in port definition or command line.

Missing ";"
Syntax error in port definition or command line.

No room on disk
There was not enough room on disk to create the keyed file. There must be room enough both for the resulting keyed file AND for a temporary file (which is about the same size as the keyed file).

Option expected
Syntax error in port definition.

Port is off screen
In a port definition, the left or top margin of the port is off the screen. The screen boundaries depend on the version of the program, which is set up for a particular machine.

Port is too high
In a port definition, the port is too high for the particular machine.

Port is too wide
In a port definition, the width is too wide for the particular machine.

Port name already used
Attempt to define a port already defined in the same layout.

Port name expected
SELECT occurred at the end of the line. The command "SELECT port-name" may not be split over two lines.

Port name not defined
The name following SELECT or ERASE was not defined in the LATEST layout.

Scrollsize is zero
In a port definition, the scroll size must not be zero.

Scrollsize too big
In a port definition, the scroll size must not be larger than the height of the port.

Too many defines
There is a limit of 16 define statements in one layout.

Unexpected end of layout
The program found a \\ command line or <<END>> before finding the end of a define statement.

Unknown command
In a command line, the program found a word that is not a valid command.

Unknown option
An option in a port definition is not a valid textport option.

Width is zero
In a port definition, the width is not allowed to be zero.

WARNING MESSAGES

Contradictory option ignored
In a port definition, the option set contained two contradictory options (such as Slow and Noslow, or Leftadjust and Centered). The second option in the set is ignored.

Extraneous lines ignored
Lines that are not part of messages are ignored, but the program flags them with a warning in case there was a missing start-of-message.

Mismatched message name
The message name in an <<END message-name>> did not match the name in the corresponding start-of-message.

Missing end-of-message
A start-of-message was not matched by an <<END>>. The end-of-message is assumed to occur at the next start-of-message or at the end of the text file.

6.D Example of Display Messages

The following is a Text Source File:

```
<<Format1>> Layout
  SetCrScheme(PtCS1);
  Define(Question, 5, 2, 35, 3, 1, [Centered]);
  Define(Reply, 50, 7, 20, 4, 2, [RightAdjust, NoTone]);
  Define(Narrate, 0, 12, 40, 9, 3, []);
\\Select(Narrate)
This is an example
of what happens
\\Left
when you display a message
from a keyed
\\Break    -- Now the program writes "--let's take a break--"
file.
\\Select(Question); Centered
This is the Question port.
\\Left
"How are you today?"
\Select(Reply)
Here is the Reply port.
\\Right
"I am fine, thank you."
\\Wait; Erase(ALL)
<<END Format1>>

<<Format2>> Layout
  SetCrScheme(PtCS1);
  Define(Question, 23, 2, 35, 3, 1, []);
  Define(Reply, 0, 8, 20, 4, 1, [NoTone]);
  Define(Narrate, 35, 16, 29, 3, 2, []);
\\Select(Narrate)
Here is the new Narrate port.
Variable string=
\\Break      -- Now the program displays the variable string.
\\Select(Question)
Notice that the Question
port has moved.
\\Select(Reply)
This is the end
of program Demo.
<<END Format2>>
```

This Text Source File is converted (with a Mulcher program) to a Keyed File called X.KFIL.

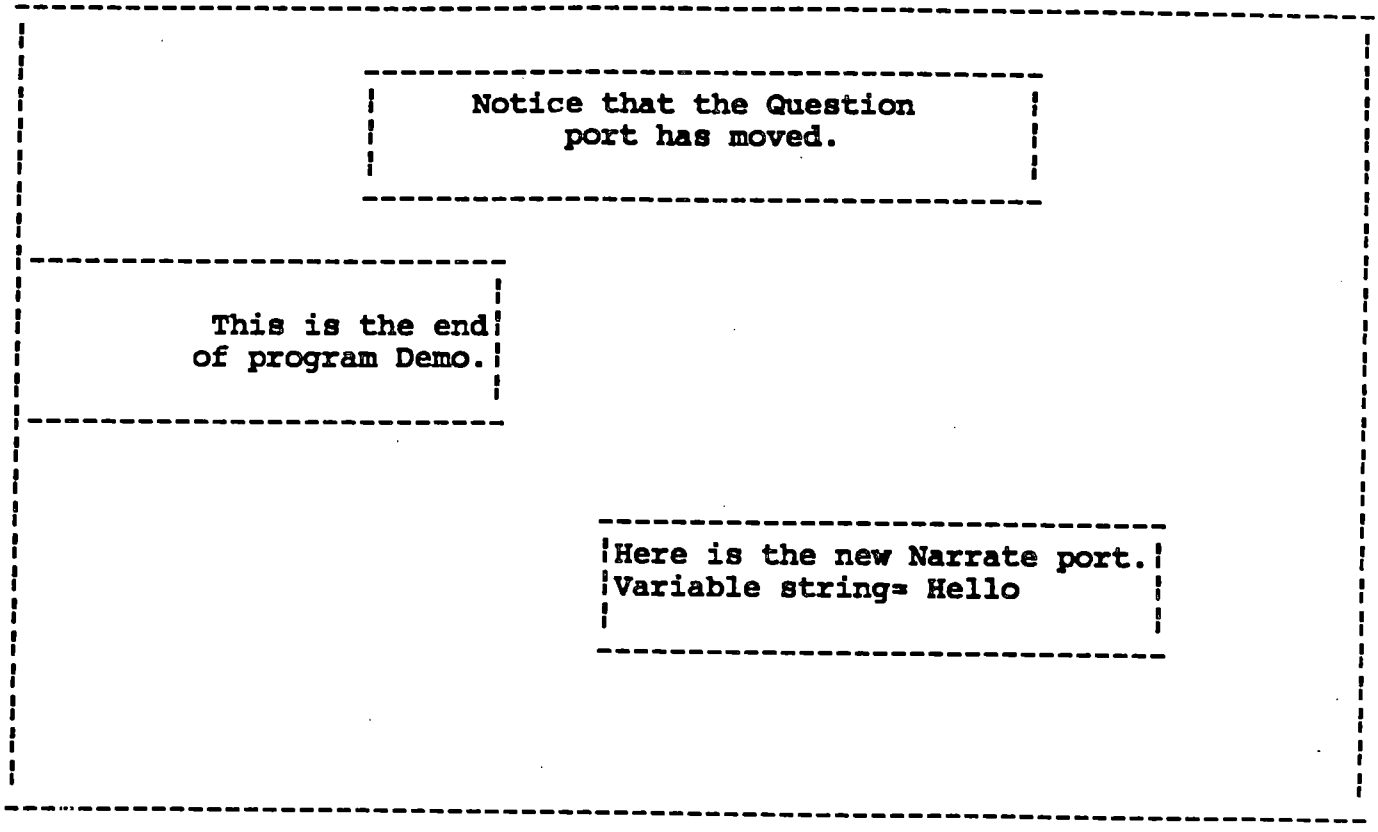
A calling program might appear:

```
PROGRAM Demo;
USES Ports, PDisplay;
CONST ...
      BlocksInMemory = 2;
      ...
BEGIN           { Program Demo }
  PtInit;
  IF KFOpen ('X.KFIL', BlocksInMemory) THEN
    BEGIN
      DispMessage('Format1');
      TxpWrite ('--let''s take a break--', [Endline]);
                        { this statement is executed during the \\BREAK }
      DispMessage('');   { return to Format1 }
      DispMessage('Format2');
      TxpWrite('Hello', []);  { this is the variable string }
      DispMessage('')
    END
  ELSE WriteLn('The message data file could not be opened!')
END.
```

This program will display. . .

```
+-----------------------------------------------------------------------+
|                                                                       |
|    +-----------------------------------+                              |
|    |    This is the Question port.     |                              |
|    |"How are you today?"               |                              |
|    +-----------------------------------+                              |
|                                                                       |
|                                          +--------------------------+ |
|                                          |Here is the Reply port.   | |
|                                          |                          | |
|                                          |     "I am fine, thank you."| |
|                                          +--------------------------+ |
|+--------------------------------------+                               |
|                          This is an example|                          |
|                             of what happens|                          |
|when you display a message                 |                          |
|from a keyed --let's take a break--        |                          |
|file.                                      |                          |
|                                           |                          |
|                                           |                          |
|                                           |                          |
|+------------------------------------------+                           |
|                                                                       |
|Please press space bar to continue:                                    |
+-----------------------------------------------------------------------+
```

Followed by:


Notice that the Question port has moved.
This is the end of program Demo.
Here is the new Narrate port.
Variable string= Hello

6.E Example of Data Messages

Data messages can be used to store a list of synonyms that can be used by the program in conjunction with the String Analysis routines. For more information on the String Analysis unit, refer to "Documentation for the StringAnalysis Unit."

The following exists in a text source file:

```
...
<<ConstantVelocity>> DATA
\ CONSTANT \ STEADY \ UNIFORM \ UNCHANG\
\ SPEED \ VELOCITY \ MOTION \
<<END>>
...
```

A calling program using the String Analysis routines might look like this:

```
...
VAR ConstantList, VelocityList : STRING;
    Answer : saLongString;
    ConstVel : INTEGER;
...
KFSelectMessage( 'ConstantVelocity');
KFGetItem( ConstantList);
  { ConstantList := '\ CONSTANT \ STEADY \ UNIFORM \ UNCHANG\' }
KFGetItem( VelocityList);
  { VelocityList := '\ SPEED \ VELOCITY \ MOTION \' }
TxpRead( Answer, 250, [Uppercase]);
saNormalize( Answer);
ConstVel:= saTwoPatterns(Answer,ConstantList,VelocityList);
...
```

7. Index

StringAnalysis Unit
Reference Guide
Version 1.0
24 July 1985

This software was developed at the Educational Technology Center of the University of California at Irvine supported, in part, by various federal grants.

Address comments or questions to

Alfred Bork or
Augusto Chioccariello
Educational Technology Center
University of California
Irvine, California 92717
(714) 856-6945

or

Stephen Franklin
Computing Facility
University of California
Irvine, California 92717
(714) 856-5154

Contributors to development of the StringAnalysis unit include J. Allen, S. Bartlett, A. Chioccarello, S. Franklin, and N. Salvador. Primary implementation was done by S. Franklin and S. Bartlett.

The StringAnalysis unit is implemented in Pascal using UCSD p-System (tm) developed at the University of California San Diego's Institute for Information System under the direction of Professor Kenneth Bowles. The UCSD p-System is now maintained and distributed by Softech Microsystems.

Table of Contents

1. Overview of Routines and Their Use

The StringAnalysis unit contains constants, types, variables, procedures and functions useful in analyzing user responses to questions posed in interactive computer based learning materials.

If the program logic requires checking or manipulating numeric values the user may have entered, the FIRST step is to isolate the portion of the input string which contains numerical data and convert the digits, decimal points and signs which represent numbers into REAL (in the Pascal sense of the word) values. The FindReal routine should be used for this task. RealToString performs the opposite conversion from REAL data into STRING data. These routines may be found in the Ports unit.

saNormalize puts a user's response into a standard form. In this form, all parts of the response which are to be considered as separate words are delimited by spaces.

ALL other routines in this unit assume that the response which is being analyzed has already been "normalized" by saNormalize.

saPatternFound and saTwoPatterns can be used in certain simple situations which occur rather frequently. Their use is not as flexible (or complex) as that of the other pattern recognition and manipulation routines in the unit.

saFindPattern is the fundamental pattern matching routine in the unit. Its capabilities generalize those of the POS function in UCSD Pascal. In many situations, however, one does not need the full flexibility of saFindPattern but can (and should) use simpler routines saPatternFound, saTwoPatterns or POS.

saDelimiters and saTokens are procedures which allow the programmer to control how saNormalize handles various punctuation characters. In order to take effect, they must be called BEFORE saNormalize.
saDelimiters is used to indicate which characters are to be considered as though they were spaces, that is, word delimiters.
saTokens is used to indicate which characters are to be considered as separate words themselves. For example, the "=" and "*" in "F=m*a" should be considered as though they were words. (A "token" is a character or string of characters which is treated as a complete word in itself.)
For most applications, however, one need not use these routines; the StringAnalysis unit default settings are usually appropriate.

saExpand is used after and in conjunction with saFindPattern to determine the maximum amount of the user's response which can be considered as matching a particular pattern.

saExtract extracts a substring from a string.

To use the StringAnalysis unit in a program, make sure that it has been

installed in *SYSTEM.LIBRARY and then included the following statement in the program:

```
USES StringAnalysis;
```

In addition to the routines described above, the StringAnalysis unit has the following data declarations:

```
CONST saMaxSize = 255;
TYPE saLongString = STRING[saMaxSize];
   { Variables for storing user responses should be of this type. }
VAR saYesWords, saNoWords: saLongString;
   { Lists of synonyms for "Yes" and "No" respectively. }
```

2. Routines

```
PROCEDURE saInit;
```

saInit must be called, ONCE and ONLY ONCE, BEFORE any call to saNormalize. It assigns the sets DelimiterSet and TokenSet (described under saNormalize) the initial default values specified in the descriptions of saDelimiters and saTokens. It also initializes StringAnalysis global variables as follows:

```
saYesWords =
   '\ YES \ YEA\ SURE\ RIGHT \ OK \ O K \ OF COURS\ YEP \ OKAY '
saNoWords  = '\ NO \ NOT \ NEGAT\ NAY \N''T \NT '
```

```
PROCEDURE saNormalize(VAR UserInput: saLongString);
```

saNormalize transforms UserInput for later processing by saFindPattern. The effect of the transformation is described by the following rules:

1) All letters are converted to upper case.
2) Each character of UserInput which is in DelimiterSet (see below) is changed into a space;
3) Each character of UserInput which is in TokenSet (see below) is surrounded by spaces;
4) A space is added to the start and one to the end of UserInput;
5) Multiple consecutive spaces are replaced by single spaces.

DelimiterSet and TokenSet are each sets of punctuation characters which are not directly accessible by the programmer. They can, however, be set by using the saDelimiters and saTokens routines.

WARNING:

saNormalize may very well INCREASE the length of the variable parameter UserInput. Make SURE that the actual string parameter you supply has room for this expansion; use the type saLongString NOT something like STRING[20].

saNormalize must be called for EACH user response which is going to be processed by any other routine in this unit.

```
PROCEDURE saDelimiters(Delimiters: STRING);
```

saDelimiters sets DelimiterSet to contain the punctuation characters which are in Delimiters and removes them from TokenSet. (DelimiterSet and TokenSet are described above under saNormalize.) The default initial value of DelimiterSet (set by saInit) is the same as the value it gets from the calling saDelimiters with

```
Delimters = '.,?;:!'.
```

As a special case, if Delimiters is the empty string (''), DelimiterSet is reassigned this default initial value. The call saDelimiters(' ') makes space the only character which is considered as a word/token delimiter.

```
PROCEDURE saTokens(Tokens: STRING);
```

saTokens sets TokenSet to contain the punctuation characters which are in Tokens and removes them from DelimiterSet.
(DelimiterSet and TokenSet are described above under saNormalize.)
The default initial value of TokenSet (set by saInit) is the same as the value it gets from the calling saTokens with
`Tokens = '+-/*=()&%$'.`
As a special case, if Tokens is the empty string (''), TokenSet is reassigned this default initial value.
The call saTokens(' ') means that no punctuation character will be considered as a token (space cannot be because it is always a token delimiter and is not even a punctuation character).

EXAMPLES of normalization:

With the default DelimiterSet and TokenSet, saNormalize will change

```
"No,   I  don't  think  e=m*(c+c) is right!  Do you?"
```

into

```
" NO I DON'T THINK E = M * ( C + C ) IS RIGHT DO YOU "
```

If, however, BEFORE saNormalized is called, the following two statements were executed:

```
saDelimiters(',*+');
saTokens('!?');
```

then saNormalize would change

```
"No,   I  don't  think  e=m*(c+c) is right!  Do you?"
```

into

```
" NO I DON'T THINK E=M (C C) IS RIGHT ! DO YOU ? "
```

```
PROCEDURE saFindPattern(Sentence, PList: saLongString; StartScan: INTEGER;
                        VAR Size, Position, WhichPattern: INTEGER);
```

saFindPattern searches Sentence, starting at position StartScan, for one of the patterns given in PList. It reports three values:

1) In Size, it returns the length of the pattern which was found.
   If no pattern was found, Size is set to 0.
2) In Position, it returns the position in Sentence where the pattern that was found started.
   If no pattern was found, Position is set to 0.
3) In WhichPattern, it reports which pattern was found by returning the position in PList of this pattern's first character.
   If no pattern was found, WhichPattern is set to 0.

PList is a Pattern List, as described in the terminology section at the start of this document.

saFindPattern assumes that Sentence has been normalized by saNormalize. Since saNormalize converts all letters to upper case, it makes no sense for any of the patterns in PList to contain any lower case letters.

saNormalize guarantees that words are always surrounded by spaces. This fact has certain consequences:

1) Any pattern which is intended to represent a full word (or words) should start and end with a space.
2) A pattern which is intended to represent the START of a word should START with a space.
3) A pattern which is intended to represent the END of a word should END with a space.

The Pattern List '\ NAY \ NO\NT ' illustrates these points in order with its 3 patterns ' NAY ', ' NO', and 'NT '.

saFindPattern reports the FIRST pattern in PList which is found in Sentence. For example, if Sentence = ' BYE MAXINE ' and PList = '# HI # MAX# BYE ', saFindPattern will find the pattern ' MAX' and return Size = 4, Position = 5, and WhichPattern = 7. Note that the first character in the pattern found is a space (' '); Position and WhichPattern give the position of this space in the strings Sentence and PList respectively where (as always) the first character of a string is in position 1.

```
FUNCTION saPatternFound(Sentence, PList: saLongString): BOOLEAN;
```

saPatternFound returns TRUE if Sentence contains one of the patterns given in the Pattern List PList and returns FALSE otherwise. It assumes that Sentence has been normalized by saNormalize.

```
FUNCTION saTwoPatterns(Sentence, PList1, PList2 : saLongString): INTEGER;
```

saTwoPatterns returns as its value an integer which reports whether Sentence contains any of the patterns given by the two Pattern Lists PList1 and PList2. The meaning of the value returned by saTwoPatterns is given as follows:

value returned	PList1 pattern found?	PList2 pattern found?
0	No	No
1	Yes	No
2	No	Yes
3	Yes	Yes

saTwoPatterns assumes that Sentence has been normalized by saNormalize.

EXAMPLE of handling "Yes/No" responses:

```
FUNCTION Affirmative: BOOLEAN;
{ Force user to respond "yes" or "no" unambiguously, returning TRUE if
  the response is "yes", FALSE if it is "no".  }
CONST MaxAnswerLength = 80;
VAR YesOrNo: INTEGER;
    Answer: saLongString;
BEGIN
   REPEAT
      TxpRead(Answer, MaxAnswerLength, []);
      saNormalize(Answer);
      YesOrNo:= saTwoPatterns(Answer, saYesWords, saNoWords);
      CASE YesOrNo OF
         0: TxpWrite('Please answer Yes or No.  ', []);
         1: TxpWrite('"Yes" it is.', []);
         2: TxpWrite('"No" answer is fine.', []);
         3: TxpWrite('Please answer Yes or No, but not both.  ', []);
         END;
   UNTIL YesOrNo IN [1,2];
   Affirmative:= 1=YesOrNo;
END;  {Affirmative}
```

```
PROCEDURE saExpand(Sentence: saLongString; VAR Start, Size: INTEGER);
```

saExpand takes the starting position (Start) and the length (Size) of a substring of Sentence and returns with Start and Size set to give the smallest substring of Sentence which includes the original substring and which starts and ends with the space character.
It assumes that Sentence has been normalized by saNormalize.
For example, if
Sentence = ' THIS IS EXACTLY WHAT I EXPECTED ';
Start = 14; Size = 6;
then the starting substring is 'TLY WH'.
saExpand will return with
Start = 9; Size = 14;
indicating that the substring has expanded to ' EXACTLY WHAT '.
If the parameters are invalid (e.g., Start < 1, Size < 1, or Start+Size > 1+LENGTH(Sentence)), saExpand will return with Start unchanged and Size=0.
Even if Sentence has not been normalized (it should have been!), saExpand stops its expansion of the original substring with Start >= 1 and Size+Start <= LENGTH(Sentence)+1.

Useful Facts:
1) Start+Size is the position of the first character AFTER the substring.
2) After using saExpand, one can replace the substring by a single space as follows: **DELETE(Sentence, Start, Size-1)**

```
PROCEDURE saExtract(Sentence: saLongString; Start, Size: INTEGER;
                    VAR Phrase: saLongString);
```

saExtract extracts from Sentence the substring starting at position Start of length Size and returns this substring in Phrase.
The substring extracted will be EMPTY ('') if ANY of the following conditions hold:
Start < 1, Size < 1, or Start+Size > 1+LENGTH(Sentence).

saExtract is similar to the built-in UCSD function COPY in that the string saExtract returns via its last parameter is actually
COPY(Sentence, Start, Size)
The programmer is advised to use saExtract and NOT USE COPY because COPY is a highly non-standard extension to Pascal (Pascal doesn't allow functions to return compound values) which has caused unpleasant and strange problems in certain situations.

3. Glossary

Delimiters - characters to be considered as though they were spaces, that is, words delimiters.

Pattern - a string of characters, usually in the context of a word, phrase or portion thereof which may be part of the user's response to a question. The StringAnalysis routines allow the empty string ('') as a legitimate pattern which is NEVER matched by any pattern. This convention is in keeping with the UCSD built-in function POS.

Pattern List - a string which is a list of patterns separated by a "separator character" which must be a punctuation character and is given by the first character of the string. For example, in the string ' NAY NON"T ', the separator character is ' and there are 3 patterns:
' NAY ', ' NO', and 'N"T '.
Multiple consecutive separators are ignored as is the last character of a Pattern List if it is the separator character. Thus, the following string has the same 3 patterns as the previous example:
'/ NAY // NO///N"T /'
If the first character is NOT a punctuation character, then the entire string is considered a single pattern.

Punctuation Character - Any printable character other than space, digits and letters.

Tokens - characters to be considered as separate words themselves.

4. Index